Exploring Complexity Science with Python



Think Complexity

O'REILLY®

Allen B. Downey

目錄

复杂性思维 中文第二版

> 来源：Think Complexity
>
> 译者：飞龙
>
> 版本：2.5
>
> 自豪地采用谷歌翻译

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- 代码仓库

赞助我

协议

KivyCN 学习资源

- Kivy 中文文档
- Think Python 中文第二版
- UCB CS61a 教材：SICP Python
- Tutorialspoint NumPy 教程

- Matplotlib 用户指南
- 斯坦福 CS229 机器学习中文讲义
- Duke STA663 计算统计学中文讲义
- 笨办法学 Python · 续
- 笨办法学 Linux
- 数据结构思维
- 写给人类的机器学习
- 计算与推断思维 中文版

一、复杂性科学

> 原文：Chapter 1 Complexity Science
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

这本书的论点是，复杂性科学是一种“新型科学”，我借鉴自 Stephen Wolfram。

2002年，Wolfram 发表了 “新科学”一文，在这里介绍了他和其他人在细胞自动机上的工作，并描述了一种用于计算系统研究的科学方法。在之后的章节中，我们会回顾 Wolfram，但是现在我打算将他的标题用于更广泛的东西。

我认为复杂性是新的，不是因为它将科学工具应用到一个新的主题，而是因为它使用不同的工具，允许不同种类的工作，并最终改变了我们认为是“科学”的东西。

为了证明差异，我将从经典科学的一个例子开始：假设有人问你为什么行星轨道是椭圆形的。你可以引用万有引力的牛顿定律，并用它来写出描述行星运动的微分方程。然后，你可以求解微分方程，并展示出解是椭圆。证明完毕！

大多数人发现这种解释令人满意。它包括一个数学推导 - 所以它有一些严格的证明 - 它解释了具体的观察，椭圆轨道，通过诉诸一般的原则，引力。

让我用另一种解释来对比一下。假设你搬到像底特律这样种族隔离的城市，你想知道为什么这样。如果你做一些研究，你可能会发现 Thomas Schelling 的一篇文章，称为“分离动态模型”，它提出了一个简单的种族隔离模型：

这里是我对这个模型的描述：

- 城市的谢林模型是一个单元格数组，每个单元格代表一个房子。这些房子被两种“智能体”占据，标有红色和蓝色，数量大致相等。大约10%的房子是空的。
- 在任何时间点，智能体可能会高兴或不高兴，这取决于附近的其他智能体。在模型的一个版本中，如果智能体至少有两个邻居像自己一样，则智能体很高兴，如果邻居是一个或者零个，则智能体不高兴。
- 这个模拟通过随机选择一个智能体来运行，并检查它是否快乐。如果是的话，没有任何反应 如果不是，智能体随机选择一个未占用的单元格并移动。

如果你从一个完全未分离的模拟城市开始，并在短时间内运行该模型，类似的智能体会聚集到一起。随着时间的流逝，这些社区会增长和合并，直到存在少量的大型社区，大多数智能体都生活在均匀的社区中。

模型中的分离程度令人惊讶，这是眞实城市的分离的解释。也许底特律是分离的，因为人们不喜欢人数太多，并且如果他们的社区的组成使他们不开心，将会搬走。

这个解释与行星运动的解释是一样的吗？许多人会说不是，但为什么？

最明显的是，谢林模型是非常抽象的，也就是说不现实的。我们很容易假设，人比行星更复杂，但是当你想想看，行星就像人一样复杂（特别是拥有人的行星）。

这两个系统都很复杂，而且这两个模型都是基于简化的；例如，在行星运动的模型中，我们包含了地球与太阳之间的力，并忽略行星之间的相互作用。

重要的区别是，对于行星运动，我们可以展示，我们忽略的力小于我们包含的力，来捍卫我们的模型。并且我们可以扩展模型，来包含其他相互作用，并显示这种效果很小。对于谢林模型，它难以合理简化。

更糟糕的是，谢林模型不符合任何物理规律，它只使用简单的计算，而不是数学推导。谢林模型不像经典科学，许多人发现它们不那么引人注目，至少一开始是这样。但是，我将尝试演示，这些模型做了大量的实用工作，包括预测，解释和设计。本书的目标之一是解释如何这样做。

1.1 范式转变

当我向人们介绍这本书时，别人经常问我，这种新型科学是不是一种范式转变。我不这么认为，并且这里是解释。

Thomas Kuhn 在 1962 年的“科学革命结构 ”中介绍了“范式转变”一词。它是指科学史上的一个过程，其中一个领域的基本假设改变，或者一个理论被另一个理论取代。他列举了哥白尼革命，燃烧的氧气模型取代了燃素说，以及相对论的出现。

复杂性科学的发展不是取代旧的模型，而是（在我看来）标准模型的逐渐转变，它们是各种种类的可接受的模型。

例如，经典模型倾向于以定律为基础，以方程式的形式表示，并通过数学推导求解。复杂性不足的模型通常是基于规则的，表示为计算，而不是由分析来模拟。

不是每个人都认为这些模型令人满意。例如，在 Sync 中，Steven Strogatz 写道了他的萤火虫自发同步模型。他展示了一个演示该现象的仿眞，但是写道：

> 对于其它随机的初始条件和其他数量的振荡器，我重复模拟了几十次。每次都会同步 [...] 现在的挑战是证明它。只有可靠的证明才能演示，同步是不可避免的，这种方式计算机都做不到；最好的证明就是澄清为什么它是不可避免的。

Strogatz 是一位数学家，所以他对证明的热情是可以理解的，但他的证明并不能解决这个现象中最有趣的部分。为了证明“同步是不可避免的”，Strogatz 做了几个简化的假设，特别是每个萤火虫可以看到所有其他的萤火虫。

在我看来，解释整个萤火虫族群为何可以同步，尽管事实上他们不能看到彼此，是更有趣的事情。这种全局行为，如何从局部交互中产生，是第（？）章的主题。这些现象的解释通常使用基于智能体的模型，它探索（以难以或不可能使用数学分析或的方式）允许或阻止同步的条件。

我是一名计算机科学家，所以我对计算模型的热情可能并不奇怪。我不是说 Strogatz 是错误的，而是人们对于提出什么问题，和用什么工具来回答他们，有不同的看法。这些意见基于价值判断，所以没有理由能够达成一致。

然而，科学家们对于哪些模型是好的科学，其他哪些是边缘科学，伪科学，或者是非科学，已经有了很大的共识。

我声称，这是本书的核心论点，即这种共识是基于时间变化的标准，复杂性科学的出现反映了这些标准的逐渐转变。

1.2 科学模型的轴线

我将经典模型描述为基于物理定律，以方程式表示，并通过数学分析求解的模型；相反，复杂系统的模型通常基于简单的规则并以计算实现。

我们可以将这一趋势看作是沿着两个轴线的转变：

基于方程式 → 基于 模拟

分析 → 计算

这种新的科学方式在其他几个方面是不同的。我在这里介绍他们，所以你知道即将会发生什么，但是在你看到本书后面的例子之前，有一些可能没有任何意义。

连续 → 离散

经典模型倾向于基于连续数学，如微积分；复杂系统的模型通常基于离散数学，包括图和细胞自动机。

线性 → 非线性

经典模型通常是线性的，或者使用非线性系统的线性近似; 复杂性科学对非线性模型更为友好。一个例子是混沌理论。

> 混沌理论在这本书中没有涉及，但是你可以在 http://en.wikipedia.org/wiki/Chaos 上阅读它。

确定性 → 随机

经典模型通常是确定性的，这可能反映了底层哲学的确定性，它在第（？）章中讨论。复杂模型往往具有随机性。

抽象 → 具体

在经典模型中，行星是质点，飞机是无摩擦的，牛是球形的（见 http://en.wikipedia.org/wiki/Spherical_cow）。像这样的简化通常对于分析是必要的，但是计算模型可能更加现实。

> 译者注：真空中的球形鸡

一，二 → 很多

在天体力学中，两体问题可以通过分析求解；而三体问题不能。经典模型通常限于少量相互作用的元素，复杂性科学作用于较大的复合体（这是名称的来源）。

单一 → 复合

在经典模型中，元素往往是可互换的；复杂模型更经常包含异质性。

这些是概括性的，所以我们不应该过于认真地对待它们。而我并不意味着弃用经典科学。更复杂的模型不一定更好；实际上通常更糟。

此外，我并不是说这些变化是突然的或完全的。相反，它们向着被认为是可接受的，值得尊重的工作的前沿逐渐迁移。过去被怀疑的工具现在很普遍，一些被广泛接受的模型现在受到审查。

例如，当 Appel 和 Haken 在 1976 年证明了四色定理时，他们使用电脑列举了 1,936 个特殊情况，在某种意义上说，这些特例是其证明的前提。当时很多数学家没有把这个定理当成真正的证明。现在计算机辅助证明是常见的，一般（但并非普遍）是可接受的。

相反，大量的经济分析基于人类行为的模型，称为“经济人”，或者一个有逼格的词：“Homo economicus”。基于这种模型的研究数十年间受到高度重视，特别是如果涉及到数学技巧的话。最近，这种模型受到怀疑，而包含不完整信息和有限理性的模型是热门话题。

1.3 一种新的的模型

复杂模型通常适用于不同的目的和解释：

预测 → 解释

谢林的分离模型可能揭示了一个复杂的社会现象，但对预测没有用。另一方面，一个简单的天体力学模型可以预测日食，在未来几年内可以精确到秒。

现实主义 → 工具主义

经典模型依赖于现实主义的解释；例如，大多数人接受电子是存在的真实事物。工具主义一种观点，即使他们假设的实体不存在，模型也可以有用。乔治·皮特写道：“所有模型都是错误的，但有些是有用的。”它可能是工具主义的座右铭。

简化论 → 整体论

简化论是一种观点，通过理解其组件来解释系统的行为。例如，元素的周期表是简化论的胜利，因为它用原子中的简单电子模型来解释元素的化学行为。整体论认为，系统层面出现的一些现象不存在于组件层面，不能在组件层面上解释。

我们在第（？）章会回到解释模型，第（？）章会回到工具主义，第（？）章会回到整体论。

1.4 一种新的工程

我一直在科学背景下谈论复杂系统，但复杂性也是工程中的变化和社会系统的组织的一个原因和影响：

中心化（集权） → 去中心化（放权）

中心化系统在概念上简单并易于分析，但去中心化系统可能更加强大。例如，万维网中的客户端向中心化服务器发送请求；如果服务器关闭，则这个服务不可用。在对等网络中，每个节点都是客户端和服务器。要取消服务，你必须删除每个 节点。

隔离 → 互动

在经典工程中，大型系统的复杂性通过隔离组件和最小化相互作用进行管理。这仍然是一个重要的工程原理；然而，廉价计算能力的普及，使得组件之间复杂交互的系统的设计变得越来越可行。

一对多 → 多对多

在许多通信系统中，广播服务正在由一些服务扩展，有时是替换。这些服务允许用户彼此通信，并创建，共享和修改内容。

自上而下 → 自下而上

在社会，政治和经济系统方面，许多通常是集中组织的活动现在都是草根运动。即使是分层结构的典范，军队，指挥和控制的也开始下放。

分析 → 计算

在经典工程中，可行的设计空间受到我们分析能力的限制。例如，设计艾菲尔铁塔成为了可能，因为 Gustave Eiffel 开发了新颖的分析技术，特别是用于处理风压负载。现在，用于计算机辅助设计和分析的工具，可以构建几乎可以想象的任何东西。弗兰克·盖里（Frank Gehry）的毕尔包古根汉美术馆（Guggenheim Museum Bilbao）是我最喜欢的例子。

设计 → 搜索

工程有时被描述为，在可行的设计空间中寻找解决方案。越来越多的搜索过程可以自动化。例如，遗传算法在大型设计空间中探索，并发现人类工程师不会想像（或喜欢）的解决方案。最终的遗传算法，演变，不可避免地生成违反人类工程规则的设计。

1.5 一种新的思维

我们现在正在深入一个领域，但是我所假设的，科学建模中的标准转变，有关 20 世纪中逻辑和认识论的发展。

亚里士多德逻辑 → 多值逻辑

在传统逻辑中，任何命题都是真或假。这个系统适用于类似数学的证明，但对于许多现实世界的应用而言是失败的（以一种戏剧化的方式）。替代方案包括多值逻辑，模糊逻辑和其他旨在处理不确定性（indeterminacy），模糊性和不确定性（uncertainty）的系统。Bart Kosko 在《模糊思维》（Fuzzy Thinking）中讨论了一些这种系统。

频率论的概率 → 贝叶斯主义

贝叶斯概率已经存在了几个世纪，但直到最近才被广泛使用，这是由于廉价计算能力变得可用，以及概率性声明中勉强接受了主观性。莎朗·贝尔奇·麦格雷恩（Sharon Bertsch McGrayne）在《不会死亡的理论》（The Theory That Would Not Die）中介绍了这一历史。

客观 → 主观

启蒙运动和现代主义哲学，建立在对客观真理的信仰上。也就是说，独立于持有他们的人的真理。20 世纪的发展，包括量子力学，哥德尔不完备定理和库恩的科学史研究，都引起了人们对“看似不可避免的主观性”的关注，甚至在“自然科学”和数学中。丽贝卡·戈德斯坦（Rebecca Goldstein）介绍了Gödel对不完备性的证明的历史背景。

物理定律 → 理论 → 模型

有些人区分了定律，理论和模型，但我认为这是一回事。使用“定律”的人很有可能认为，它在客观上是真实的，不可改变的；使用“理论”的人承认它可以修改；而“模型”承认它是基于简化和近似的。

一些被称为“物理定律”的概念是真正的定义；实际上，其他的只是模型的断言，它很好预测或解释了系统的行为。我们在第（？）章中会回到屋里定律的本质。

确定性 → 不确定性

确定性是一个观点，所有事件都是由之前事件导致，不可避免。不确定性的形式包括随机性，概率因果和基本不确定性。我们在第（？）章再回到这个主题。

这些趋势并不普遍或完整，但核心观点正沿着这些轴线转变。作为证据，考虑对托马斯·库恩（Thomas Kuhn）的《科学革命的结构》（The Structure of Scientific Revolutions）的反应，公布后受到谴责，现在被认为几乎毫无争议。

这些趋势是复杂性科学的因和果。例如，高度抽象的模型现在更容易接受，因为人们预期，每个系统都应该有一个独特的，正确的模型。相反，复杂系统的发展挑战了确定性，和物理定律的相关概念。

本章概述了本书中出现的主题，但在看到示例之前，并不是全部都是有意义的。当你读到本书的最后，你可能会发现，再次阅读本章会有帮助。

二、图

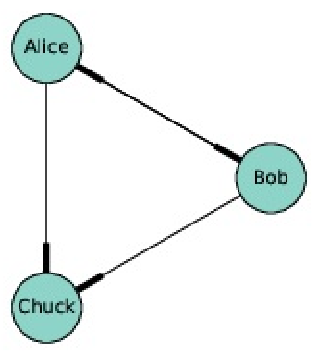
> 原文：Chapter 2 Graphs
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

本书的前三章有关一些模型，它们描述了由组件和组件之间的连接组成的系统。例如，在生态食物网中，组件是物种，连接代表捕食者和猎物的关系。

在本章中，我介绍了 NetworkX，一个用于构建和研究这些模型的 Python 包。我们从 Erdős-Rényi 模型开始，它具有一些有趣的数学属性。在下一章中，我们将介绍更有用的，解释现实系统的模型。

本章的代码在本书仓库中的`chap02.ipynb`中。使用代码的更多信息请参见第（？）章。

2.1 图是什么？



> 图 2.1：表示社交网络的有向图

对于大多数人来说，图是数据集的视觉表示，如条形图或股票价格对于时间的绘图。这不是本章的内容。

在本章中，图是一个系统的表示，它包含离散的互连元素。元素由节点表示，互连由边表示。

例如，你可以表示一个路线图，每个城市都是一个节点，每个城市之间的路线是一条边。或者你可以表示一个社交网络，每个人是节点，如果他们是朋友，两个人之间有边，否则没有。

在某些图中，边具有长度，成本或权重等属性。例如，在路线图中，边的长度可能代表两个城市之间的距离，或旅行时间。在社交网络中，可能会有不同的边来表示不同种类的关系：朋友，商业伙伴等。

边可以是有向或无向的，这取决于它们表示的关系是不对称的还是对称的。在路线图中，你可能会使用有向边表示单向街道，使用无向边表示双向街道。在某些社交网络，如 Facebook，好友是对称的：如果 A 是 B 的朋友，那么 B 也是 A 的朋友。但在 Twitter 上，“关注”关系并不对称；如果 A 关注了 B，这并不意味着 B 关注 A。因此，你可以使用无向边来表示 Facebook 网络，并将有向边用于 Twitter。

图具有有趣的数学属性，并且有一个称为图论的数学分支，用于研究它们。

图也很有用，因为有许多现实世界的问题可以使用图的算法来解决。例如，Dijkstra 的最短路径算法，是从图中找到某个节点到所有其他节点的最短路径的有效方式。路径是两个节点之间的，带有边的节点序列。

图的节点通常以圆形或方形绘制，边通常以直线绘制。例如，上面的有向图中，节点可能代表在 Twitter 上彼此“关注”的三个人。线的较厚部分表示边的方向。在这个例子中，爱丽丝和鲍勃相互关注，都关注查克，但查克没有关注任何人。

下面的无向图展示了美国东北部的四个城市；边上的标签表示驾驶时间，以小时为单位。在这个例子中，节点的位置大致对应于城市的地理位置，但是通常图的布局是任意的。

2.2 NetworkX

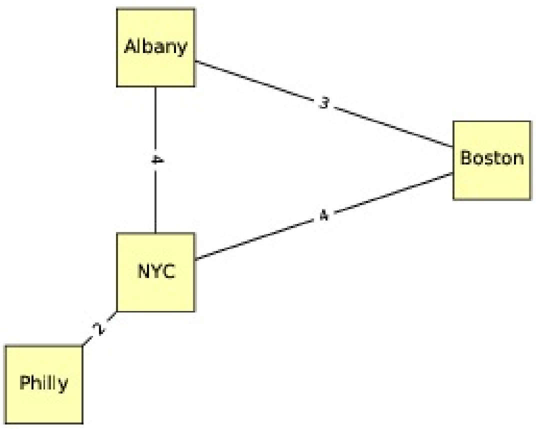


图 2.2：表示城市和高速公路的无向图

为了表示图，我们将使用一个名为 NetworkX 的包，它是 Python 中最常用的网络库。你可以在 https://networkx.github.io/ 上阅读更多信息，但是我们之后会解释。

我们可以通过导入 NetworkX 和实例化 `nx.DiGraph` 来创建有向图：

```
import networkx as nx
G = nx.DiGraph()
```

通常将 NetworkX 导入为 `nx`。此时，`G` 是一个 `DiGraph` 对象，不包含节点和边。我们可以使用 `add_node` 方法添加节点：

```
G.add_node('Alice')
G.add_node('Bob')
G.add_node('Chuck')
```

现在我们可以使用 `nodes` 方法获取节点列表：

```
>>> G.nodes()
['Alice', 'Bob', 'Chuck']
```

添加边的方式几乎相同：

```
G.add_edge('Alice', 'Bob')
G.add_edge('Alice', 'Chuck')
G.add_edge('Bob', 'Alice')
G.add_edge('Bob', 'Chuck')
```

我们可以使用 `edges` 来获取边的列表：

```
>>> G.edges()
[('Alice', 'Bob'), ('Alice', 'Chuck'),
 ('Bob', 'Alice'), ('Bob', 'Chuck')]
```

NetworkX 提供了几个绘图的功能；`draw_circular` 将节点排列成一个圆，并使用边将它们连接：

```
nx.draw_circular(G,
                 node_color=COLORS[0],
                 node_size=2000,
                 with_labels=True)
```

这就是我用来生成图（？）的代码。`with_labels` 选项标注了节点；在下一个例子中，我们将看到如何标注边。

为了产生图（？），我们以一个字典开始，它将每个城市的名称，映射为对应的经纬度：

```
pos = dict(Albany=(-74, 43),
           Boston=(-71, 42),
           NYC=(-74, 41),
           Philly=(-75, 40))
```

因为这是个无向图，我实例化了 `nx.Graph` ：

```
G = nx.Graph()
```

之后我可以使用 `add_nodes_from` 来迭代 `pos` 的键，并将它们添加为节点。

```
G.add_nodes_from(pos)
```

下面我会创建一个字典，将每条边映射为对应的驾驶时间。

```
drive_times = {('Albany', 'Boston'): 3,
               ('Albany', 'NYC'): 4,
               ('Boston', 'NYC'): 4,
               ('NYC', 'Philly'): 2}
```

现在我可以使用 `add_edges_from` ，它迭代了 `drive_times` 的键，并将它们添加为边：

```
G.add_edges_from(drive_times)
```

现在我不使用 `draw_circular` ，它将节点排列成一个圆圈，而是使用 `draw` ，它接受 `pos` 作为第二个参数：

```
nx.draw(G, pos,
        node_color=COLORS[1],
        node_shape='s',
        node_size=2500,
        with_labels=True)
```

`pos` 是一个字典，将每个城市映射为其坐标； `draw` 使用它来确定节点的位置。

要添加边的标签，我们使用 `draw_networkx_edge_labels` ：

```
nx.draw_networkx_edge_labels(G, pos,
                             edge_labels=drive_times)
```

`drive_times` 是一个字典，将每条边映射为它们之间的驾驶距离，每条边表示为城市名称的偶对。这就是我生成图（？）的方式。

在这两个例子中，这些节点是字符串，但是通常它们可以是任何可哈希的类型。

2.3 随机图

随机图就像它的名字一样：一个随机生成的节点和边的图。当然，有许多随机过程可以生成图，所以有许多种类的随机图。

其中一个更有趣的是 Erdős-Rényi 模型，PaulErdős 和 AlfrédRényi 在 20 世纪 60 年代研究过它。

Erdős-Rényi 图（ER 图）的特征在于两个参数：`n` 是节点的数量，`p` 是任何两个节点之间存在边的概率。见 http://en.wikipedia.org/wiki/Erdos-Renyi_model。

Erdős 和 Rényi 研究了这些随机图的属性；其令人惊奇的结果之一就是，随着随机的边被添加，随机图的属性会突然变化。

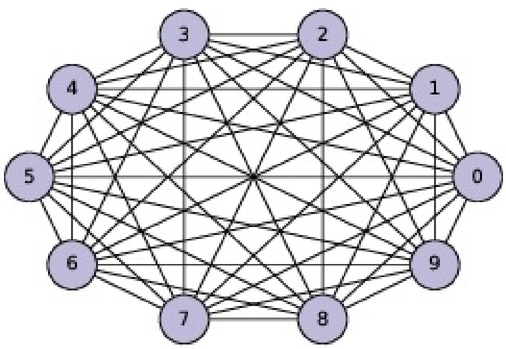
展示这类转变的一个属性是连通性。如果每个节点到每个其他节点都存在路径，那么无向图是连通的。

在 ER 图中，当 `p` 较小时，图是连通图的概率非常低，而 `p` 较大时接近 `1`。在这两种状态之间，在 `p` 的特定值处存在快速转变，表示为 `p*`。

Erdős 和 Rényi 表明，这个临界值是 `p* = lnn / n`，其中 `n` 是节点数。如果 `p < p*`，随机图 `G(n, p)` 不太可能连通，并且如果 `p > p*`，则很可能连通。

为了测试这个说法，我们将开发算法来生成随机图，并检查它们是否连通。

2.4 生成图



我将首先生成一个完全图，这是一个图，其中每个节点都彼此连接。

这是一个生成器函数，它接收节点列表并枚举所有不同的偶对。如果你不熟悉生成器函数，你可能需要阅读附录？，然后回来。

```
def all_pairs(nodes):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i>j:
                yield u, v
```

你可以使用 `all_pairs` 来构造一个完全图。

```
def make_complete_graph(n):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(all_pairs(nodes))
    return G
```

`make_complete_graph` 接受节点数 `n` ，并返回一个新的 `Graph` ，拥有 `n` 个节点，所有节点之间都有边。

以下代码生成了一个包含 10 个节点的完全图，并绘制出来。

```
complete = make_complete_graph(10)
nx.draw_circular(complete,
                 node_color=COLORS[2],
                 node_size=1000,
                 with_labels=True)
```

图（？）显示了结果。不久之后，我们将修改此代码来生成 ER 图，但首先我们将开发函数来检查图是否是连通的。

2.5 连通图

如果每个节点到每个其他节点都存在路径，这个图就是连通图。请见 http://en.wikipedia.org/wiki/Connectivity_(graph_theory)。

对于许多涉及图的应用，检查图是否连通是很有用的。幸运的是，有一个简单的算法。

你可以从任何节点起步，并检查是否可以到达所有其他节点。如果你可以到达一个节点 `v` ，你可以到达 `v` 的任何一个邻居，他们是 `v` 通过边连接的任何节点。

`Graph` 类提供了一个称为 `neighbors` 的方法，返回给定节点的邻居列表。例如，在上一节中我们生成的完全图中：

```
>>> complete.neighbors(0)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

假设我们从节点 `s` 起步。我们可以将 `s` 标记为“已访问”，然后我们可以标记它的邻居。然后我们标记邻居的邻居，依此类推，直到你无法再到达任何节点。如果访问了所有节点，则图是连通图。

以下是 Python 中的样子：

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

`reachable_nodes` 接受 `Graph` 和起始节点 `start`，并返回可以从 `start` 到达的节点集合，他们。

最初，已访问的集合是空的，我们创建一个名为 `stack` 的列表，跟踪我们发现但尚未处理的节点。最开始，栈包含单个节点 `start`。

现在，每次在循环中，我们：

- 从栈中删除一个节点。
- 如果节点已在 `seen` 中，我们返回到步骤 1。
- 否则，我们将节点添加到 `seen`，并将其邻居添加到栈。

当栈为空时，我们无法再到达任何节点，所以我们终止了循环并返回。

例如，我们可以找到从节点 `0` 可到达的，完全图中的所有节点：

```
>>> reachable_nodes(complete, 0)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

最初，栈包含节点 `0`，`seen` 是空的。第一次循环中，节点 `0` 添加到了 `seen`，所有其他节点添加到了栈中（因为它们都是节点 `0` 的邻居）。

下一次循环中，`pop` 返回栈中的最后一个元素，即节点 `9.` 因此，节点 `9` 被添加到 `seen`，并且其邻居被添加到栈。

请注意，同一个节点在栈中可能会出现多次；实际上，具有 `k` 个邻居的节点将添加到栈 `k` 次。稍后我们将寻找方法，来使此算法更有效率。

我们可以使用 `reachable_nodes` 来编写 `is_connected`：

```
def is_connected(G):
    start = next(G.nodes_iter())
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

`is_connected`通过调用`nodes_iter`来选择一个起始节点，`node_iter`返回一个迭代器对象，并将结果传递给`next`，返回第一个节点。

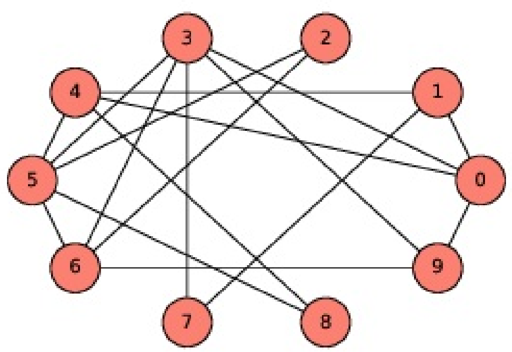
`reachable`获取了一组节点，它们可以从`start`到达。如果这个集合的大小与图的大小相同，那意味着我们可以访问所有节点，也就是这个图是连通的。

一个完全图是连通的，毫不奇怪：

```
>>> is_connected(complete)
True
```

下一节中，我们会生成 ER 图，并检查它们是否是连通的。

2.6 生成 ER图



> 图 2.4：ER 图，`n=10`，`p=0.3`

ER 图`G(n, p)`包含`n`个节点，每对节点以概率为`p`的边连接。生成 ER 图类似于生成完全图。

以下生成器函数枚举所有可能的边，并使用辅助函数`flip`，来选择哪些应添加到图中：

```
def random_pairs(nodes, p):
    for i, u in enumerate(nodes):
        for j, v in enumerate(nodes):
            if i>j and flip(p):
                yield u, v
```

`flip`以给定概率`p`返回`True`，以互补的概率`1-p`返回`False`。

```
from numpy.random import random

def flip(p):
    return random() < p
```

最后，`make_random_graph`生成并返回 ER 图`G(n, p)`。

```
def make_random_graph(n, p):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(random_pairs(nodes, p))
    return G
```

`make_random_graph`几乎和`make_complete_graph`，唯一的不同是它使用`random_pairs`而不是`all_pairs`。

这里是`p=0.3`的例子：

```
random_graph = make_random_graph(10, 0.3)
```

图（？）展示了结果。这个图是连通图；事实上，大多数`p=10`并且`p=3`的 ER 图都是连通图。在下一节中，我们将看看有多少。

2.7 连通性的概率

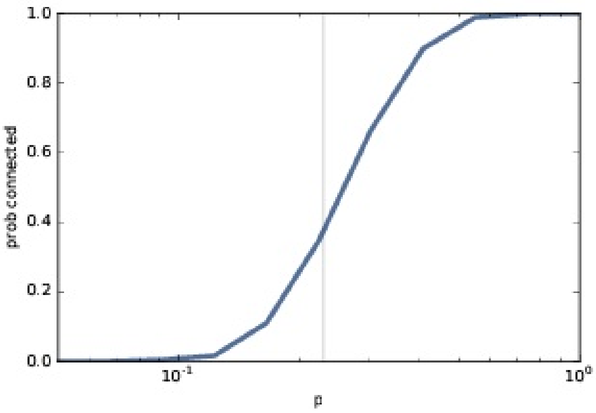


图 2.5：连通性的概率，`n=10`，`p`是一个范围。竖直的线展示了预测的临界值。

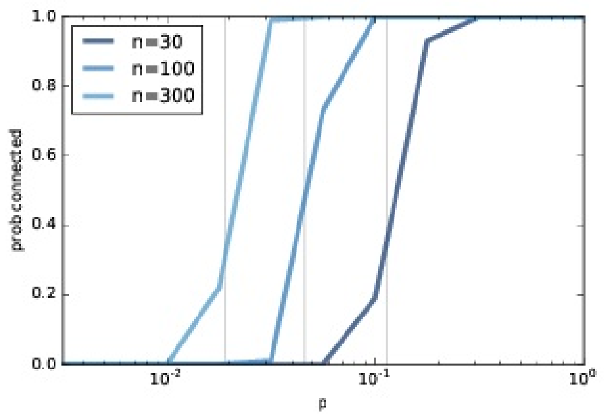


图 2.6：连通性的概率，`n` 是多个值，`p` 是一个范围。

对于 `n` 和 `p` 的给定值，我们想知道 `G(n, p)` 连通的概率。我们可以通过生成大量随机图，来计算有多少个来估计它。就是这样：

```
def prob_connected(n, p, iters=100):
    count = 0
    for i in range(iters):
        random_graph = make_random_graph(n, p)
        if is_connected(random_graph):
            count += 1
    return count/iters
```

`iters` 是我们生成的随机图的数量。随着我们增加 `iter`，估计的概率就会更加准确。

```
>>> prob_connected(10, 0.3, iters=10000)
0.6454
```

在具有这些参数的 10000 个 ER 图中，6498 个是连通的，因此我们估计其中65%是连通的。所以 0.3 接近临界值，在这里连通概率从接近 0 变为接近 1。根据 Erdős 和 Rényi，`p* = lnn / n = 0.23`。

我们可以通过估计一系列 `p` 值的连通概率，来更清楚地了解转变：

```
import numpy as np

n = 10
ps = np.logspace(-2.5, 0, 11)
ys = [prob_connected(n, p) for p in ps]
```

这是我们看到的使用 NumPy 的第一个例子。按照惯例，我将 NumPy 导入为 `np`。函数 `logspace` 返回从 `10 ** -2.5` 到 `10 ** 0 = 1` 的 11 个元素的数组，在对数刻度上等间隔。

为了计算 `y`，我使用列表推导来迭代 `ps` 的元素，并计算出每个值为 `p` 的随机图的连通概率。

图（？）展示了结果，竖直的线为 `p*` 。从 0 到 1 的转变发生在预测的临界值附近。在对数刻度上，这个转变大致对称。

对于较大的 `n` 值，图（？）展示了类似的结果。随着 `n` 的增加，临界值越来越小，转变越来越突然。

这些实验与 Erdős 和 Rényi 在其论文中证明的结果一致。

2.8 图的算法分析

这一章中，我提出了一个检查图是否连通的算法；在接下来的几章中，我们将再次看到更多的图的算法。并且我们要分析这些算法的性能，了解它们的运行时间如何随着图大小的增加而增长。

如果你还不熟悉算法分析，在你继续之前，你应该阅读附录一。

图算法的增长级别通常表示为顶点数量 `n` ，以及边数量 `m` 的函数。

作为一个例子，我们分析从前面的 `reachable_nodes` ：

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

每次循环，我们从栈中弹出一个节点；默认情况下， `pop` 删除并返回列表的最后一个元素，这是一个常数时间的操作。

接下来我们检查节点是否被已访问，这是一个集合，所以检查成员是常数时间。

如果节点还没有访问，我们添加它是常量时间，然后将邻居添加到栈中，这相对于邻居数量是线性的。

为了使用 `n` 和 `m` 表达运行时间，我们可以将每个节点添加到 `seen` 和 `stack` 的总次数加起来。

每个节点只添加一次，所以添加的总数为 `n` 。

但是节点可能多次被添加到栈，具体取决于它们有多少邻居。如果节点具有 `k` 个邻居，则它会被添加到栈 `k` 次。当然，如果它有 `k` 个邻居，那意味着它拥有 `k` 个边。

所以添加到栈的总数是边的数量 `m` 的两倍，由于我们考虑每个边两次。

因此，这个函数的增长级别为 `O(n + m)` ，我们可以说，即运行时间与 `n` 或 `m` 成正比，以较大者为准。

如果我们知道 `n` 和 `m` 之间的关系，我们可以简化这个表达式。例如，在完全图中，边数是 `n(n-1)/ 2` ，它属于 `O(n^2)` 。所以对于一个完全图， `reachable_nodes` 是二次于 `n` 的。

2.9 练习

本章的代码在 `chap02.ipynb` 中，它是本书的仓库中的 Jupyter 笔记本。使用此代码的更多信息，请参阅第（？）节。

练习 1：启动 `chap02.ipynb` 并运行代码。笔记本中嵌入了一些简单的练习，你可能想尝试一下。

练习 2：我们分析了 `reachable_nodes` 的性能，并将其分类为 `O(n + m)` ，其中 `n` 是节点数， `m` 是边数。继续分析， `is_connected` 的增长级别是什么？

```
def is_connected(G):
    start = next(G.nodes_iter())
    reachable = reachable_nodes(G, start)
    return len(reachable) == len(G)
```

练习 3 ：在我实现 `reachable_nodes` 时，你可能很困惑，因为向栈中添加所有邻居而不检查它们是否已访问，明显是低效的。编写一个该函数的版本，在将邻居添加到栈之前检查它们。这个“优化”是否改变了增长级别？它是否使函数更快？

> 译者注：在弹出节点时将其添加到 `seen` ，在遍历邻居时检查它们是否已访问。

练习 4：

实际上有两种 ER 图。我们在本章中生成的一种， `G(n，p)` 的特征是两个参数，节点数量和节点之间的边的概率。

一种替代定义表示为 `G(n，m)` ，也以两个参数为特征：节点数 `n` 和边数 `m` 。在这个定义中，边数是固定的，但它们的位置是随机的。

使用这个替代定义，重复这一章的实验。这里是几个如何处理它的建议：

1. 编写一个名为 `m_pairs` 的函数，该函数接受节点列表和边数 `m` ，并返回随机选择的 `m` 个边。一个简单的方法是，生成所有可能的边的列表，并使用 `random.sample` 。
2. 编写一个名为 `make_m_graph` 的函数，接受 `n` 和 `m` ，并返回 `n` 个节点和 `m` 个边的随机图。
3. 创建一个 `prob_connected` 的版本，使用 `make_m_graph` 而不是 `make_random_graph` 。

4. 计算一系列 `m` 值的连通概率。

与第一类 ER 图的结果相比，该实验的结果如何？

三、小世界图

> 原文：Chapter 3 Small world graphs
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

现实世界中的许多网络，包括社交网络在内，具有“小世界属性”，即节点之间的平均距离，以最短路径上的边数来衡量，远远小于预期。

在本章中，我介绍了斯坦利·米拉格（Stanley Milgram）的著名的“小世界实验”，这是小世界属性在真正的社交网络中的第一次科学演示。之后我们将考虑 Watts-Strogatz 图，它是一个小世界图的模型。我将复制 Watts 和 Strogatz 所做的实验，并解释它打算展示的东西。

这个过程中，我们将看到两种新的图算法：广度优先搜索（BFS）和 Dijkstra 算法，用于计算图中节点之间的最短路径。

本章的代码在本书仓库的 `chap03.ipynb` 中。使用代码的更多信息请参见第（？）章。

3.1 Stanley Milgram

斯坦利·米拉格（Stanley Milgram）是美国社会心理学家，他进行了两项最著名的社会科学实验，即 Milgram 实验，研究人们对权威的服从（http://en.wikipedia.org/wiki/Milgram_experiment）和小世界实验，研究了社交网络的结构（http://en.wikipedia.org/wiki/Small_world_phenomenon）。

在小世界实验中，Milgram 向堪萨斯州威奇托（Wichita, Kansas）的几个随机选择的人发送了包裹，带有一个指示，要求他们向马萨诸塞州沙龙（Sharon, Massachusetts）的目标人员发送一封附带的信（在我长大的地方，波士顿附近），目标人员通过名字和职业确定。受访者被告知，只有当他亲自认识目标人员时，才可以将该信直接邮寄给目标；否则他们按照指示，将信和同一个指示发送给他们认为的，更有可能认识目标人员的亲戚或朋友。

许多信件从来没有发出过，但是对于发出的信件，平均路径长度（信件转发次数）的大约为 6。这个结果用于确认以前的观察（和猜测），社交网络中任何两个人之间的通常距离是“六度分隔”。

这个结论令人惊讶，因为大多数人都希望社交网络本地化 - 人们往往会靠近他们的朋友 - 而且在一个具有本地连接的图中，路径长度往往会与地理距离成比例增加。例如，我的大多数朋友都住在附近，所以我猜想社交网络中节点之间的平均距离是大约 50 英里。威奇托距离波士

顿约有 1600 英里，所以如果 Milgram 的信件穿过了社交网络的典型环节，那么他们应该有 32 跳，而不是 6 跳。

3.2 Watts 和 Strogatz

1998年，Duncan Watts 和 Steven Strogatz 在 Nature 杂志上发表了一篇“小世界网络的集体动态”（Collective dynamics of ’small-world’ networks）论文，提出了小世界现象的解释。 你可以从 http://www.nature.com/nature/journal/v393/n6684/abs/393440a0.html 下载。

Watts 和 Strogatz 从两种很好理解的图开始：随机图和正则图。在随机图中，节点随机连接。在正则图中，每个节点具有相同数量的邻居。他们考虑这些图的两个属性，群聚性和路径长度：

群聚是图表的“集团性”（cliquishness）的度量。在图中，集团是所有节点的子集，它们彼此连接；在一个社交网络中，集团是一群人，彼此都是朋友。Watts 和 Strogatz 定义了一个群聚系数，用于量化两个节点彼此连接，并同时连接到同一个节点的可能性。

路径长度是两个节点之间的平均距离的度量，对应于社交网络中的分离度。

Watts 和 Strogatz 表明，正则图具有高群聚性和长路径长度，而大小相同的随机图通常具有群聚性和短路径长度。所以这些都不是一个很好的社交网络模型，它是高群聚性与短路径长度的组合。

他们的目标是创造一个社交网络的生成模型。生成模型通过为构建或导致现象的过程建模，试图解释现象。Watts 和 Strogatz 提出了用于构建小世界图的过程：

1. 从一个正则图开始，节点为 `n`，每个节点连接 `k` 个邻居。
2. 选择边的子集，并将它们替换为随机的边来“重新布线”。

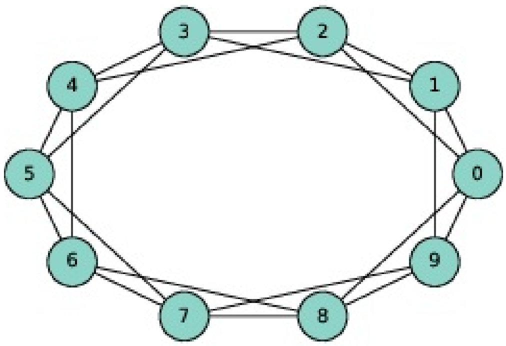
边的重新布线的概率是参数 `p`，它控制图的随机性。当 `p = 0` 时，该图是正则的；`p = 1` 是随机的。

Watts 和 Strogatz 发现，较小的 `p` 值产生高群聚性的图，如正则图，短路径长度的图，如随机图。

在本章中，我将按以下步骤复制 Watts 和 Strogatz 实验：

- 我们将从构建一个环格（ring lattice）开始，这是一种正则图。
- 然后我们和 Watts 和 Strogatz 一样重新布线。
- 我们将编写一个函数来测量群聚度，并使用 NetworkX 函数来计算路径长度。
- 然后，我们为范围内的 `p` 值计算群聚度和路径长度。
- 最后，我将介绍一种用于计算最短路径的高效算法，Dijkstra 算法。

3.3 环格



> 图 3.1 `n=10`，`k=4` 的环格

正则图是每个节点具有相同数量的邻居的图；邻居的数量也称为节点的度。环格是一种正则图，Watts 和 Strogatz 将其用作模型的基础。在具有 `n` 个节点的环格中，节点可以排列成圆形，每个节点连接 `k` 个最近邻居。

例如，`n = 3` 和 `k = 2` 的环形网格将拥有以下边：`(0, 1)`，`(1, 2)`，`(2, 0)`。请注意，边从编号最高的节点“绕回”0。

更一般地，我们可以像这样枚举边：

```
def adjacent_edges(nodes, halfk):
    n = len(nodes)
    for i, u in enumerate(nodes):
        for j in range(i+1, i+halfk+1):
            v = nodes[j % n]
            yield u, v
```

`adjacent_edges` 接受节点列表和参数 `halfk`，它是 `k` 的一半。它是一个生成器函数，一次产生一个边。它使用模运算符 `%`，从编号最高的节点绕回最低的节点。

我们可以这样测试：

```
>>> nodes = range(3)
>>> for edge in adjacent_edges(nodes, 1):
...     print(edge)
(0, 1)
(1, 2)
(2, 0)
```

现在我们可以使用 `adjacent_edges` 来生成环格。

```
def make_ring_lattice(n, k):
    G = nx.Graph()
    nodes = range(n)
    G.add_nodes_from(nodes)
    G.add_edges_from(adjacent_edges(nodes, k//2))
    return G
```

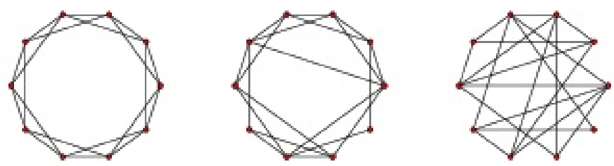
注意，`make_ring_lattice`使用地板除计算`halfk`，所以如果`k`是奇数，它将向下取整并产生具有度`k-1`的环格。这可能不是我们想要的，但现在还不错。

我们可以像这样测试函数：

```
lattice = make_ring_lattice(10, 4)
```

图（？）展示了结果。

3.4 WS 图



> 图 3.2 WS 图，`n=20`，`k=4`，`p=0`（左边），`p=0.2`（中间），`p=1`（右边）。

为了制作 Watts-Strogatz（WS）图，我们从一个环格开始，并为一些边“重新布线”。 在他们的论文中，Watts 和 Strogatz 以特定顺序考虑边，并用概率`p`重新布置每个边。 如果边被重新布置，则它们使第一个节点保持不变，并随机选择第二个节点。它们不允许自环或多边；也就是说，节点不能拥有到它自身的边，并且两个节点之间不能拥有多个边。

这是我的这个过程的实现。

```
def rewire(G, p):
    nodes = set(G.nodes())
    for edge in G.edges():
        if flip(p):
            u, v = edge
            choices = nodes - {u} - set(G[u])
            new_v = choice(tuple(choices))
            G.remove_edge(u, v)
            G.add_edge(u, new_v)
```

参数 `p` 是边的重新布线的概率。 `for` 循环枚举了边，并使用 `flip` ，它以概率 `p` 返回 `True` ，来选择哪些被重新布置。

如果我们重新布置节点 `u` 到节点 `v` 的边，我们必须选择一个节点来替换 `v` ，称为 `new_v` 。为了计算可能的选择，我们从节点集开始，它是一个集合，并且移除 `u` 和它的邻居，这避免了自环和多边。

然后我们从选项中选择new_v，将 `u` 到 `v` 的现有删除，并从添加一个 `u` 到 `new_v` 的新边。

另外，表达式 `G[u]` 返回一个字典，他的键是包含 `u` 的邻居。在这种情况下，它比使用 `G.neighbors` 更快一点。

这个函数不按照 Watts 和 Strogatz 指定的顺序考虑边缘，但它似乎不会影响结果。

图（？）展示了 `n = 20` ， `k = 4` 和范围内 `p` 值的 WS 图。当 `p = 0` 时，该图是环格。当 `p = 1` 时，它是完全随机的。我们将看到，有趣的事情发生在两者之间。

3.5 群聚性

下一步是计算群聚系数，它量化了节点形成集团的趋势。 集团是一组完全连接的节点；也就是说，在集团中的所有节点对之间都存在边。

假设一个特定的节点 `u` 具有 `k` 个邻居。如果所有的邻居都相互连接，则会有 `k(k-1)/2` 个边。 实际存在的这些边的比例是 `u` 的局部群聚系数，表示为 `Cu` 。它被称为“系数”，因为它总是在 0 和 1 之间。

如果我们计算所有节点上的 `Cu` 平均值，我们得到“网络平均群聚系数”，表示为 `C` 。

这是一个计算它的函数。

```
def node_clustering(G, u):
    neighbors = G[u]
    k = len(neighbors)
    if k < 2:
        return 0

    total = k * (k-1) / 2
    exist = 0
    for v, w in all_pairs(neighbors):
        if G.has_edge(v, w):
            exist +=1
    return exist / total
```

同样，我使用 `G [u]` ，它返回一个字典，键是节点的邻居。如果节点的邻居少于两个，则群聚系数未定义，但为简便起见， `node_clustering` 返回 0。

否则，我们计算邻居之间的可能的边数量， `total` ，然后计算实际存在的边数量。结果是存在的所有边的比例。

我们可以这样测试函数：

```
>>> lattice = make_ring_lattice(10, 4)
>>> node_clustering(lattice, 1)
0.5
```

在 `k=4` 的环格中，每个节点的群聚系数是 `0.5` （如果你不相信，可以看看图（？））。

现在我们可以像这样计算网络平均群聚系数：

```
def clustering_coefficient(G):
    cc = np.mean([node_clustering(G, node) for node in G])
    return cc
```

`np.mean` 是个 NumPy 函数，计算列表或数组中元素的均值。

然后我们可以像这样测试：

```
>>> clustering_coefficient(lattice)
0.5
```

这个图中，所有节点的局部群聚系数是 0.5，所以节点的平均值是 0.5。当然，我们期望这个值和 WS 图不同。

3.6 最短路径长度

下一步是计算特征路径长度 `L` ，它是每对节点之间最短路径的平均长度。 为了计算它，我将从 NetworkX 提供的函数开始， `shortest_path_length` 。 我会用它来复制 Watts 和 Strogatz 实验，然后我将解释它的工作原理。

这是一个函数，它接受图并返回最短路径长度列表，每对节点一个。

```
def path_lengths(G):
    length_map = nx.shortest_path_length(G)
    lengths = [length_map[u][v] for u, v in all_pairs(G)]
    return lengths
```

`nx.shortest_path_length` 的返回值是字典的字典。外层字典每个节点 `u` 到内层字典的映射，内层字典是每个节点 `v` 到 `u->v` 的最短路径长度的映射。

使用来自 `path_lengths` 的长度列表，我们可以像这样计算 `L` ：

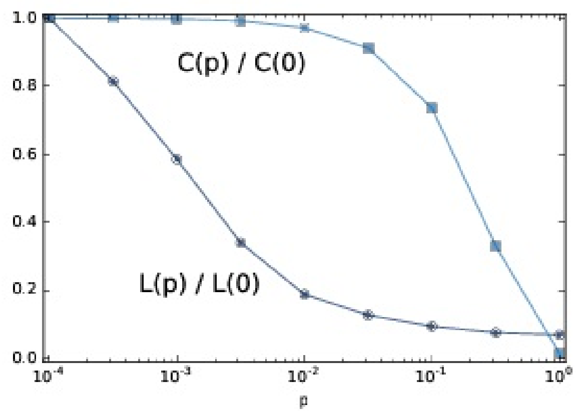
```
def characteristic_path_length(G):
    return np.mean(path_lengths(G))
```

并且我们可以使用小型的环格来测试它。

```
>>> lattice = make_ring_lattice(3, 2)
>>> characteristic_path_length(lattice)
1.0
```

这个例子中，所有三个节点都互相连接，所以平均长度为 1。

3.7 WS 实验



> 图 3.3：WS 图的群聚系数 `C` 和特征路径长度 `L`，其中 `n=1000, k=10`，`p` 是一个范围。

现在我们准备复制 WS 实验，它表明对于一系列 `p` 值，WS 图具有像正则图像那样的高群聚性，像随机图一样的短路径长度。

我将从 `run_one_graph` 开始，它接受 `n`，`k` 和 `p`；它生成具有给定参数的 WS图，并计算平均路径长度 `mpl` 和群聚系数 `cc`：

```
def run_one_graph(n, k, p):
    ws = make_ws_graph(n, k, p)
    mpl = characteristic_path_length(ws)
    cc = clustering_coefficient(ws)
    print(mpl, cc)
    return mpl, cc
```

Watts 和 Strogatz 用 `n = 1000` 和 `k = 10` 进行实验。使用这些参数，`run_one_graph` 在我的电脑上需要大约一秒钟；大部分时间用于计算平均路径长度。

现在我们需要为范围内的 `p` 计算这些值。我将再次使用 NumPy 函数 `logspace` 来计算 `ps`：

```
ps = np.logspace(-4, 0, 9)
```

对于每个`p`的值，我生成了 3 个随机图，并且我们将结果平均。这里是运行实验的函数：

```
def run_experiment(ps, n=1000, k=10, iters=3):
    res = {}
    for p in ps:
        print(p)
        res[p] = []
        for _ in range(iters):
            res[p].append(run_one_graph(n, k, p))
    return res
```

结果是个字典，将每个`p`值映射为`(mpl, cc)`偶对的列表。

最后一步就是聚合结果：

```
L = []
C = []
for p, t in sorted(res.items()):
    mpls, ccs = zip(*t)
    mpl = np.mean(mpls)
    cc = np.mean(ccs)
    L.append(mpl)
    C.append(cc)
```

每次循环时，我们取得一个`p`值和一个`(mpl, cc)`偶对的列表。我们使用`zip`来提取两个列表，`mpls`和`ccs`，然后计算它们的均值并将它们添加到`L`和`C`，这是路径长度和群聚系数的列表。

为了在相同的轴上绘制`L`和`C`，我们通过除以第一个元素，将它们标准化：

```
L = np.array(L) / L[0]
C = np.array(C) / C[0]
```

图（？）展示了结果。随着`p`的增加，平均路径长度迅速下降，因为即使少量随机重新布线的边，也提供了图区域之间的捷径，它们在格中相距很远。另一方面，删除局部链接降低了群聚系数，但是要慢得多。

因此，存在较宽范围的`p`，其中 WS 图具有小世界图的性质，高群聚度和短路径长度。

这就是为什么 Watts 和 Strogatz 提出了 WS 图，作为展示小世界现象的，现实世界网络的模型。

3.8 能有什么解释？

如果你问我，为什么行星轨道是椭圆形的，我最开始会为一个行星和一个恒星建模；我将在 http://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation 上查找万有引力定律，并用它为行星的运动写出一个微分方程。之后我会扩展轨道方程式，或者更有可能在 http://en.wikipedia.org/wiki/Orbit_equation 上查找。通过一个小的代数运算，我可以得出产生椭圆轨道的条件。之后我会证明我们看做行星的物体满足这些条件。

人们，或至少是科学家，一般对这种解释感到满意。它有吸引力的原因之一是，模型中的假设和近似值似乎是合理的。行星和恒星不是真正的质点，但它们之间的距离是如此之大，以至于它们的实际尺寸可以忽略不计。同一太阳系中的行星可以影响彼此的轨道，但效果通常较小。而且我们忽视相对论的影响，再次假定它们很小。

这也因为它是基于方程式的。我们可以用闭式表达轨道方程，这意味着我们可以有效地计算轨道。这也意味着我们可以得出轨道速度，轨道周期和其他数量的一般表达式。

最后，我认为这是因为它具有数学证明的形式。它从一组公理开始，通过逻辑和分析得出结果。但重要的是要记住，证明属于模型，而不是现实世界。也就是说，我们可以证明，行星的理想模型产生一个椭圆轨道，但是我们不能证明这个模型与实际的行星有关（实际上它不是）。

- 这些模型可以做什么工作：它们是预测性的还是说明性的，还是都有？
- 这些模型的解释，是否比基于更传统模型的解释更不满意？为什么？
- 我们应该如何刻画这些和更传统的模型之间的差异？他们在种类还是程度上不同？

在这本书中，我将提供我对这些问题的回答，但它们是暂时性的，有时是投机性的。我鼓励你怀疑地思考他们，并得出你自己的结论。

3.9 广度优先搜索

当我们计算最短路径时，我们使用了 NetworkX 提供的一个函数，但是我没有解释它是如何工作的。为此，我将从广度优先搜索开始，这是用于计算最短路径的 Dijkstra 算法的基础。

在第（？）节，我提出了 `reachable_nodes` ，它寻找从给定的起始节点可以到达的所有节点：

```
def reachable_nodes(G, start):
    seen = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in seen:
            seen.add(node)
            stack.extend(G.neighbors(node))
    return seen
```

我当时没有这么说，但它执行深度优先搜索（DFS）。现在我们将修改它来执行广度优先搜索（BFS）。

为了了解区别，想象一下你正在探索一座城堡。你最开始在一个房间里，带有三个门，标记为 A，B 和 C 。你打开门 C 并发现另一个房间，它的门被标记为 D ，E 和 F。

下面你打开哪个门呢？如果你打算冒险，你可能想更深入城堡，选择 D，E 或 F。这是一个深度优先搜索。

但是，如果你想更系统化，你可以在 D，E 和 F 之前回去探索 A 和 B，这将是一个广度优先搜索。

在 `reachable_nodes` 中，我们使用 `list.pop` 选择下一个节点来“探索”。默认情况下， `pop` 返回列表的最后一个元素，这是我们添加的最后一个元素。在这个例子中，这是门 F。

如果我们要执行 BFS，最简单的解决方案是将第一个元素从栈中弹出：

```
node = stack.pop(0)
```

这有效，但速度很慢。在 Python 中，弹出列表的最后一个元素需要常数时间，但是弹出第一个元素线性于列表的长度。在最坏的情况下，就是堆栈的长度 `O(n)` ，这使得 BFS 的 `O(nm)` 的实现比 `O(n + m)` 差得多。

我们可以用双向队列（也称为 `deque` ）来解决这个问题。 `deque` 的一个重要特征就是，你可以在开头和末尾添加和删除元素。要了解如何实现，请参阅 https://en.wikipedia.org/wiki/Double-ended_queue 。

Python 在 `collections` 模块中提供了 `deque` ，所以我们可以像这样导入它：

```
from collections import deque
```

我们可以使用它来编写高效的 BFS：

```
def reachable_nodes_bfs(G, start):
    seen = set()
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node not in seen:
            seen.add(node)
            queue.extend(G.neighbors(node))
    return seen
```

差异在于：

- 我用名为 `queue` 的 `deque` 替换了名为 `stack` 的列表。
- 我用 `popleft` 替换 `pop` ，它删除并返回队列的最左边的元素，这是第一个添加的元素。

这个版本恢复为 `O(n + m)` 。现在我们做好了寻找最短路径的准备。

3.10 （简化的）Dijkstra 算法

Edsger W. Dijkstra 是荷兰计算机科学家，发明了一种有效的最短路径算法（参见 http://en.wikipedia.org/wiki/Dijkstra's_algorithm）。他还发明了信号量，它是一种数据结构，用于协调彼此通信的程序（参见 http://en.wikipedia.org/wiki/Semaphore_(programming)和 Downey，《The Little Book of Semaphores》）。

作为一系列计算机科学论文的作者，Dijkstra 是着名（臭名昭着）的。有些比如“反对 GOTO 语句的案例”（A Case against the GO TO Statement），对编程实践产生了深远的影响。其他比如“真正的计算机科学教学的残酷”（On the Cruelty of Really Teaching Computing Science）的人，很有娱乐性，但效果却不好。

Dijkstra 算法解决了“单源最短路径问题”，这意味着它寻找从给定的“源”节点到图中每个其他节点（或至少每个连接节点）的最小距离。

我们最开始考虑算法的简化版本，所有边的长度相同。更一般的版本适用于任何非负的边的长度。

简化版本类似于第一节中的广度优先搜索 除了我们用称为 `dist` 的字典替换集合 `seen`，该字典将每个节点映射为与源的距离：

```
def shortest_path_dijkstra(G, start):
    dist = {start: 0}
    queue = deque([start])
    while queue:
        node = queue.popleft()
        new_dist = dist[node] + 1

        neighbors = set(G[node]) - set(dist)
        for n in neighbors:
            dist[n] = new_dist

        queue.extend(neighbors)
    return dist
```

这是它的工作原理：

- 最初，队列包含单个元素 `start`，`dist` 将 `start` 映射为距离 0（这是 `start` 到自身的距离）。
- 每次循环中，我们使用 `popleft` 获取节点，按照添加到队列的顺序。
- 接下来，我们发现节点的所有邻居都没有在 `dist` 中。
- 由于从起点到节点的距离是 `dist [node]`，到任何未访问的邻居的距离是 `dist [node] +1`。
- 对于每个邻居，我们向 `dist` 添加一个条目，然后将邻居添加到队列中。

只有在我们使用 BFS 而不是 DFS 时，这个算法才有效。为什么？

第一次循环中，`node` 是 `start` ，`new_dist` 为 `1` 。所以 `start` 的邻居距离为 1，并且进入了队列。

当我们处理 `start` 的邻居时，他们的所有邻居距离为 `2` 。我们知道，他们中没有一个距离为 `1` ，因为如果有的话，我们会在第一次迭代中发现它们。

类似地，当我们处理距离为 2 的节点时，我们将他们的邻居的距离设为 `3` 。我们知道它们中没有一个的距离为 `1` 或 `2` ，因为如果有的话，我们将在之前的迭代中发现它们。

等等。如果你熟悉归纳证明，你可以看到这是怎么回事。

但是，在我们开始处理距离为 `2` 的节点之前，只有我们处理了距离为 `1` 的所有节点，这个论证才有效，依此类推。这正是 BFS 所做的。

在本章末尾的练习中，你将使用 DFS 编写 Dijkstra 算法的一个版本，以便你有机会看到出现什么问题。

3.11 练习

练习 1：

在一个环格中，每个节点的邻居数量相同。邻居的数量称为节点的度，所有节点的度相同的图称为正则图。

所有环格都是正则的，但不是所有的正则图都是环格。特别地，如果 `k` 是奇数，则不能构造环格，但是我们可以构建一个正则图。

编写一个名为 `make_regular_graph` 的函数，该函数接受 `n` 和 `k` ，并返回包含 `n` 个节点的正则图，其中每个节点都有 `k` 个邻居。如果不可能使用 `n` 和 `k` 的给定值来制作正则图，则该函数应该抛出 `ValueError` 。

练习 2：

我的 `reachable_nodes_bfs` 实现是有效的，因为它是 `O(n + m)` 的，但它产生了很多开销，将节点添加到队列中并将其删除。 NetworkX 提供了一个简单，快速的 BFS 实现，可从 GitHub 上的 NetworkX 仓库获取，网址为 https://github.com/networkx/networkx/blob/master/networkx/algorithms/components/connected.py。

这里是我修改的一个版本，返回一组节点：

```
def _plain_bfs(G, source):
    seen = set()
    nextlevel = {source}
    while nextlevel:
        thislevel = nextlevel
        nextlevel = set()
        for v in thislevel:
            if v not in seen:
                seen.add(v)
                nextlevel.update(G[v])
    return seen
```

将这个函数与 `reachable_nodes_bfs` 相比，看看哪个更快。之后看看你是否可以修改这个函数来实现更快的 `shortest_path_dijkstra` 版本。

练习 3：

下面的 BFS 实现包含两个性能错误。它们是什么？这个算法的实际增长级别是什么？

```
def bfs(top_node, visit):
    """Breadth-first search on a graph, starting at top_node."""
    visited = set()
    queue = [top_node]
    while len(queue):
        curr_node = queue.pop(0)    # Dequeue
        visit(curr_node)            # Visit the node
        visited.add(curr_node)

        # Enqueue non-visited and non-enqueued children
        queue.extend(c for c in curr_node.children
                     if c not in visited and c not in queue)
```

练习 4：在第（？）节中，我说了除非使用 BFS，Dijkstra 算法不能工作。编写一个 `shortest_path_dijkstra` 的版本，它使用 DFS，并使用一些例子测试它，看看哪里不对。

练习 5：

Watts 和 Strogatz 的论文的一个自然问题是，小世界现象是否特定于它的生成模型，或者其他类似模型是否产生相同的定性结果（高群聚和短路径长度）。

为了回答这个问题，选择 WS 模型的一个变体并重复实验。 你可能会考虑两种变体：

- 不从常规图开始，从另一个高群聚的图开始。 例如，你可以将节点放置在二维空间中的随机位置，并将每个节点连接到其最近的 `k` 个邻居。
- 尝试不同种类的重新布线。

如果一系列类似的模型产生类似的行为，我们认为论文的结果是可靠的。

练习 6：

Dijkstra 算法解决了“单源最短路径”问题，但为了计算图的特征路径长度，我们其实需要解决“多源最短路径”问题。

当然，一个选择是运行 Dijkstra 算法 `n` 次，每个起始节点一次。 对于某些应用，这可能够好，但是有更有效的替代方案。

找到一个多源最短路径的算法并实现它。请参阅 https://en.wikipedia.org/wiki/Shortest_path_problem#All-pairs_shortest_paths。

将实现的运行时间与运行 Dijkstra 算法 `n` 次进行比较。哪种算法在理论上更好？哪个在实践中更好？NetworkX 使用了哪一个？

四、无标度网络

> 原文：Chapter 4 Scale-free networks
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在本章中，我们将处理来自在线社交网络的数据，并使用 WS 图对其进行建模。WS 模型像数据一样，具有小世界网络的特点，但是与数据不同，它的节点到节点的邻居数目变化很小。

这种差异是 Barabási 和 Albert 开发的网络模型的动机。BA 模型捕捉到邻居数量的观察到的变化，它具有小的世界属性之一，短路径长度，但它没有一个小世界网络的高聚类。

本章最后讨论了 WS 和 BA 图，作为小世界网络的解释模型。

本章的代码位于本书的仓库中的 `chap04.ipynb` 中。使用代码的更多信息，请参见第（？）章。

4.1 社交网络数据

WS 图的目的是，模拟自然科学和社会科学中的网络。Watts 和 Strogatz 在他们最初的论文中，查看了电影演员的网络（如果他们出现在同一部电影中，就是连接的）。美国西部的电网；和 C. elegans 线虫脑中的神经元网络 。他们发现，所有这些网络都具有小世界图的高群聚性和短路径长度特征。

在本节中，我们将使用不同的数据集，Facebook 用户及其朋友的数据集，来进行相同的分析。如果你对 Facebook 不熟悉，那么彼此连接的用户被称为“朋友”，而不管他们在现实世界中的关系的性质如何。

我将使用来自斯坦福网络分析项目（SNAP）的数据，该项目分享了来自在线社交网络和其他来源的大型数据集。具体来说，我将使用他们的 Facebook 数据集 [1]，其中包括 4039 个用户和 88,234 个朋友关系。该数据集位于本书的仓库中，但也可以从 SNAP 网站上获取。

> [1] J. McAuley and J. Leskovec. Learning to Discover Social Circles in Ego Networks. NIPS, 2012.

数据文件为每条边包含一行，用户由 0 到 4038 之间的整数标识。下面是读取文件的代码：

```
def read_graph(filename):
    G = nx.Graph()
    array = np.loadtxt(filename, dtype=int)
    G.add_edges_from(array)
    return G
```

NumPy 提供了函数 `loadtext` ，它读取给定的文件，并以 NumPy 数组的形式返回内容。参数 `dtype` 指定数组元素的类型。

然后我们可以使用 `add_edges_from` 迭代数组的行，并创建边。结果如下：

```
>>> fb = read_graph('facebook_combined.txt.gz')
>>> n = len(fb)
>>> m = len(fb.edges())
>>> n, m
(4039, 88234)
```

节点和边的数量与数据集的文档一致。

现在我们可以检查这个数据集是否具有小世界图的特征：高群聚性和短路径长度。

第（？）节中，我们编写了一个函数，来计算网络平均群聚系数。NetworkX 提供了一个叫做的函数 `average_clustering` ，它可以更快地完成相同的工作。

但是对于更大的图，它们都太慢，需要与 `nk^2` 成正比的时间，其中 `n` 是节点数， `k` 是每个节点的邻居数。

幸运的是， `NetworkX` 提供了一个通过随机抽样来估计群聚系数的函数。你可以像这样调用它：

```
    from networkx.algorithms.approximation import average_clustering
    average_clustering(G, trials=1000)
```

下面函数对路径长度做了类似的事情：

```
def random_path_lengths(G, nodes=None, trials=1000):
    if nodes is None:
        nodes = G.nodes()
    else:
        nodes = list(nodes)

    pairs = np.random.choice(nodes, (trials, 2))
    lengths = [nx.shortest_path_length(G, *pair)
               for pair in pairs]
    return lengths
```

`G` 是一个图， `nodes` 是节点列表，我们应该从中抽样， `trials` 是要抽样的随机路径的数量。如果 `nodes` 是 `None` ，我们从整个图表中进行抽样。

`pairs` 是随机选择的节点的 NumPy 数组，对于每个采样有一行两列。

列表推导式枚举数组中的行，并计算每对节点之间的最短距离。结果是路径长度的列表。

`estimate_path_length` 生成一个随机路径长度列表，并返回它们的平均值：

```
def estimate_path_length(G, nodes=None, trials=1000):
    return np.mean(random_path_lengths(G, nodes, trials))
```

我会使用 `average_clustering` 来计算 `C` ：

```
C = average_clustering(fb)
```

并使用 `estimate_path_lengths` 来计算 `L` ：

```
L = estimate_path_lengths(fb)
```

群聚系数约为 `0.61` ，这是较高的，正如我们所期望的那样，如果这个网络具有小世界特性。

平均路径为 `3.7` ，在 4000 多个用户的网络中相当短。毕竟这是一个小世界。

现在让我们看看是否可以构建一个 WS 图，与此网络具有相同特征。

4.2 WS 模型

在 Facebook 数据集中，每个节点的平均边数约为 22。由于每条边都连接到两个节点，度的均值是每个节点边数的两倍：

```
>>> k = int(round(2*m/n))
>>> k
44
```

我们可以用 `n=4039` 和 `k=44` 创建一个 WS 图。 `p=0` 时，我们会得到一个环格。

```
lattice = nx.watts_strogatz_graph(n, k, 0)
```

在这个图中，群聚较高： `C` 是 0.73，而在数据集中是 0.61。但是 `L` 为 46，远远高于数据集！

使用 `p=1` 我们得到一个随机图：

```
random_graph = nx.watts_strogatz_graph(n, k, 1)
```

在随机图中，`L` 是 2.6，甚至比数据集（3.7）短，但 `C` 只有 0.011，所以这是不好的。

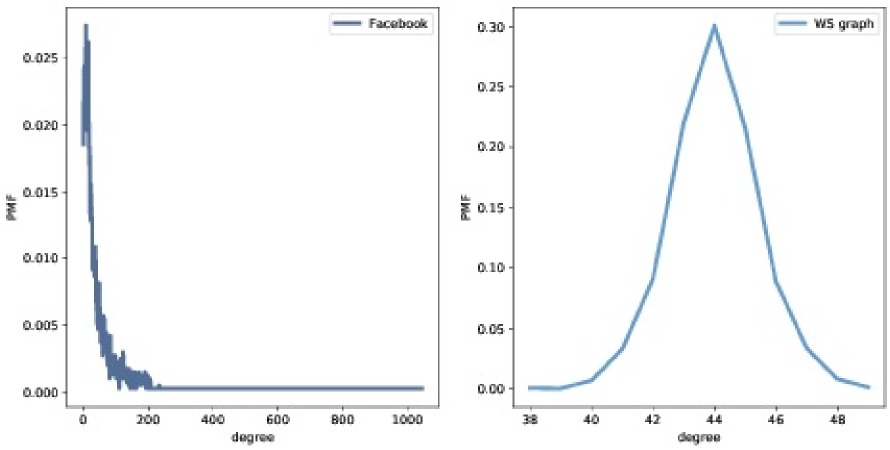
通过反复试验，我们发现，当 `p=0.05` 时，我们得到一个高群聚和短路径长度的 WS 图：

```
ws = nx.watts_strogatz_graph(n, k, 0.05, seed=15)
```

在这个图中 `C` 是 `0.63`，比数据集高一点，`L` 是 3.2，比数据集低一点。所以这个图很好地模拟了数据集的小世界特征。

到现在为止还不错。

4.3 度



> 图 4.1：Facebook 数据集和 WS 模型中的度的 PMF。

回想一下，节点的度是它连接到的邻居的数量。如果 WS 图是 Facebook 网络的一个很好的模型，它应该具有相同的总（或平均）度，理想情况下不同节点的度数相同。

这个函数返回图中的度的列表，每个节点对应一项：

```
def degrees(G):
    return [G.degree(u) for u in G]
```

数据集中的度的均值是 43.7；WS 模型中的度的均值是 44。到目前为止还不错。

但是，WS 模型中的度的标准差为 1.5；数据中的标准差是 52.4。有点糟。

这里发生了什么？为了更好地查看，我们必须看看度的 分布，而不仅仅是均值和标准差。

我将用一个 Pmf 对象来表示度的分布，它在 `thinkstats2` 模块中定义。Pmf 代表“概率质量函数”；如果你不熟悉这个概念，你可以阅读 Think Stats 第二版的第三章，网址是 http://greenteapress.com/thinkstats2/html/thinkstats2004.html 。

简而言之，Pmf 是值到概率的映射。Pmf 是每个可能的度 `d` ，到度为 `d` 的节点比例的映射。

作为一个例子，我将构建一个图，拥有节点 `1, 2, 3` ，连接到中心节点 `0` ：

```
G = nx.Graph()
G.add_edge(1, 0)
G.add_edge(2, 0)
G.add_edge(3, 0)
nx.draw(G)
```

这里是图中的度的列表：

```
>>> degrees(G)
[3, 1, 1, 1]
```

节点 `0` 度为 3，其它度为 1。现在我可以生成一个 Pmf，它表示这个度的分布：

```
>>> from thinkstats2 import Pmf
>>> Pmf(degrees(G))
Pmf({1: 0.75, 3: 0.25})
```

产生的 `Pmf` 是一个对象，将每个度映射到一个比例或概率。在这个例子中，75％的节点度为 1，25％度为 3。

现在我们生成一个 `Pmf` ，包含来自数据集的节点的度，并计算均值和标准差：

```
>>> pmf_ws = Pmf(degrees(ws))
>>> pmf_ws.mean(), pmf_ws.std()
(44.000, 1.465)
```

我们可以使用 `thinkplot` 模块来绘制结果：

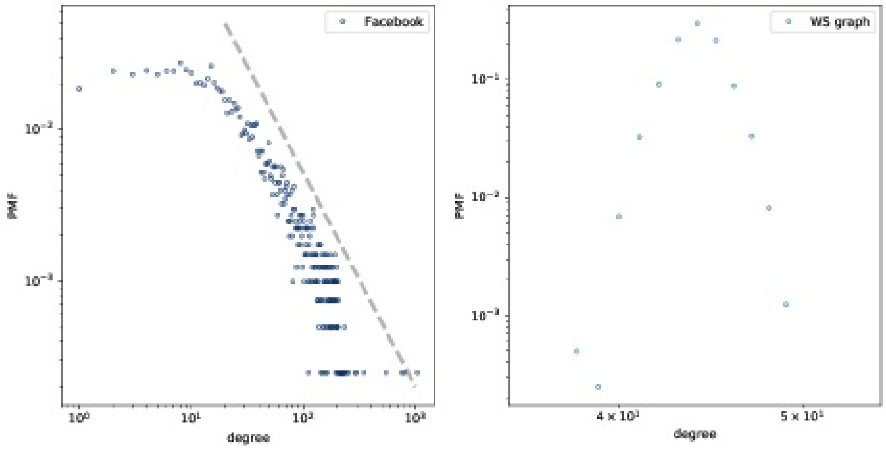
```
thinkplot.Pdf(pmf_fb, label='Facebook')
thinkplot.Pdf(pmf_ws, label='WS graph')
```

图（？）显示了这两个分布。他们是非常不同的。

在 WS 模型中，大多数用户有大约 44 个朋友；最小值是 38，最大值是 50。这个变化不大。在数据集中，有很多用户只有 1 或 2 个朋友，但有一个人有 1000 多个！

像这样的分布，有许多小的值和一些非常大的值，被称为重尾。

4.4 重尾分布



> 图 4.2：Facebook 数据集和 WS 模型中的度的 PMF，在双对数刻度下。

在复杂性科学的许多领域中，重尾分布是一个常见特征，它们将成为本书的一个反复出现的主题。

我们可以在双对数轴绘制它，来获得重尾分布的更清晰的图像，就像上面那副图那样。这种转换突显了分布的尾巴；也就是较大值的概率。

在这种转换下，数据大致在一条直线上，这表明分布的最大值与概率之间存在“幂律”关系。在数学上，

```
PMF(k) ~ k^(−α)
```

其中 `PMF(k)` 是度为 `k` 的节点的比例，`α` 是一个参数，符号 `~` 表示当 `k` 增加时，PMF 渐近于 `k^(−α)` 。

如果我们把对两边取对数，我们得到：

```
logPMF(k) ~ −α logk
```

因此，如果一个分布遵循幂律，并且我们在双对数刻度上绘制 `PMF(k)` 与 `k` 的关系，那么我们预计至少对于 `k` 的较大值，将有一条斜率为 `-α` 的直线。

所有的幂律分布都是重尾的，但是还有其他重尾分布不符合幂律。我们将很快看到更多的例子。

但首先，我们有一个问题：WS 模型拥有高群聚性和短路径长度，我们在数据中也看到了，但度的分布根本不像数据。这种差异就启发了我们下一个主题，Barabási-Albert 模型。

4.5 Barabási-Albert 模型

1999 年，Barabási 和 Albert 发表了一篇论文“随机网络中的标度的出现”（Emergence of Scaling in Random Networks），描述了几个现实世界的网络的结构特征，包含一些图，它们展示了电影演员，万维网（WWW）页面和美国西部电网设施的互联性。你可以从 http://www.sciencemag.org/content/286/5439/509 下载该论文。

他们测量每个节点的度并计算 `PMF(k)`，即节点度为 `k` 的比例。然后他们在双对数标度上绘制 `PMF(k)` 与 `k` 的关系。这些曲线可用一条直线拟合，至少对于 `k` 的较大数值；所以他们得出结论，这些分布是重尾的。

他们还提出了一个模型，生成了属性相同的图。模型的基本特征与 WS 模型不同，它们是：

增长：

BA 模型不是从固定数量的顶点开始，而是从一个较小图开始，每次添加一个顶点。

优先连接：

当创建一个新的边时，它更可能连接到一个已经有很多边的节点。这种“富者更富”的效应是一些现实世界网络增长模式的特征。

最后，他们表明，由 Barabási-Albert（BA）模型模型生成的图，度的分布遵循幂律。

具有这个属性的图有时被称为无标度网络，原因我不会解释；如果你好奇，可以在 http://en.wikipedia.org/wiki/Scale-free_network 上阅读更多内容。

NetworkX 提供了一个生成 BA 图的函数。我们将首先使用它；然后我会告诉你它的工作原理。

```
ba = nx.barabasi_albert_graph(n=4039, k=22)
```

参数是 `n` 要生成的节点数量，`k` 是每个节点添加到图形时的起始边数。我选择 `k=22`，是因为这是数据集中每个节点的平均边数。

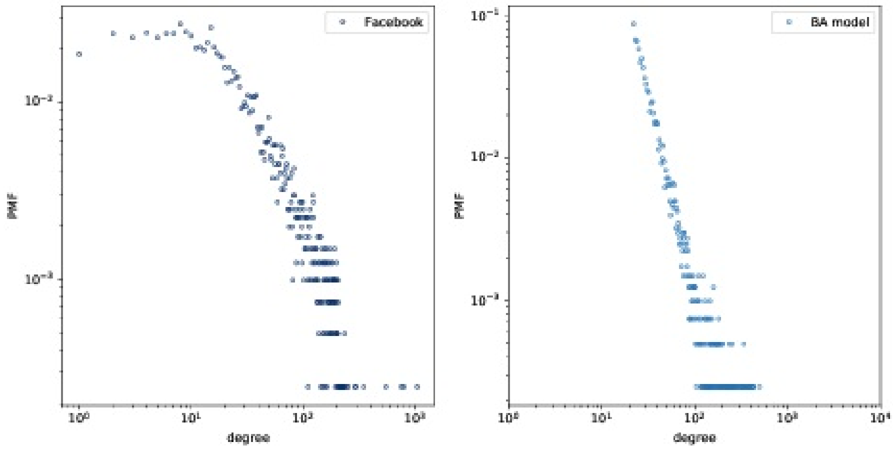


图 4.3：Facebook 数据集和 BA 模型中的节点的 PMF，在双对数刻度上。

所得图形拥有 4039 个节点，每个节点有 21.9 个边。由于每条边连接两个节点，度的均值为 43.8，非常接近数据集中的度的均值 43.7。

度的标准差为 40.9，略低于数据集 52.4，但比我们从 WS 图得到的数值好 1.5 倍。

图（？）以双对数刻度展示了 Facebook 网络和 BA 模型的度的分布。模型并不完美；特别 `k` 是在小于 10 时偏离了数据。但尾巴看起来像是一条直线，这表明这个过程产生了遵循幂律的度的分布。

所以在重现度的分布时，BA 模型比 WS 模型更好。但它有小世界的属性？

在这个例子中，平均路径长度 `L` 是 2.5，这比实际的网络的 `L = 3.69` 更小。所以这很好，虽然可能太好了。

另一方面，群聚系数 `C` 为 0.037，并不接近数据集中的值 0.61。所以这是一个问题。

下表总结了这些结果。WS 模型捕获了小世界的特点，但没有度的分布。BA 模型捕获了度的分布，和平均路径长度，至少是近似的，但没有群聚系数。

在本章最后的练习中，你可以探索其他可以捕获所有这些特征的模型。

	Facebook	WS 模型	BA 模型
`C`	0.61	0.63	0.037
`L`	3.69	3.23	2.51
度的均值	43.7	44	43.7
度的标准差	52.4	1.5	40.1
幂律？	可能	不是	是

表 4.1：与两个模型相比，Facebook 网络的特征。

4.6 生成 BA 图

在前面的章节中，我们使用了 NetworkX 函数来生成BA图。现在让我们看看它的工作原理。这是一个 `barabasi_albert_graph` 的版本，我做了一些更改，使其更易于阅读：

```
def barabasi_albert_graph(n, k):

    G = nx.empty_graph(k)
    targets = list(range(k))
    repeated_nodes = []

    for source in range(k, n):

        G.add_edges_from(zip([source]*k, targets))

        repeated_nodes.extend(targets)
        repeated_nodes.extend([source] * k)

        targets = _random_subset(repeated_nodes, k)

    return G
```

`n` 是我们想要的节点的数量，`k` 是每个新节点边的数量（近似为每个节点的边的数量）。

我们从一个 `k` 个节点和没有边的图开始。然后我们初始化两个变量：

`targets` ：

`k` 个节点的列表，它们将被连接到下一个节点。最初 `targets` 包含原来的 `k` 个节点；之后它将包含节点的随机子集。

`repeated_nodes` ：

一个现有节点的列表，如果一个节点有 `k` 条边，那么它出现 `k` 次。当我们从 `repeated_nodes` 选择时，选择任何节点的概率与它所具有的边数成正比。

每次循环中，我们添加源节点到 `targets` 中的节点的边。然后我们更新 `repeated_nodes` ，通过添加每个目标一次，以及新的节点 `k` 次。

最后，我们选择节点的子集作为下一次迭代的目标。以下是 `_random_subset` 的定义：

```
def _random_subset(repeated_nodes, k):
    targets = set()
    while len(targets) < k:
        x = random.choice(repeated_nodes)
        targets.add(x)
    return targets
```

每次循环中， `_random_subset` 从 `repeated_nodes` 选择，并将所选节点添加到 `targets` 。因为 `targets` 是一个集合，它会自动丢弃重复项，所以只有当我们选择了 `k` 不同的节点时，循环才会退出。

4.7 累积分布函数（CDF）

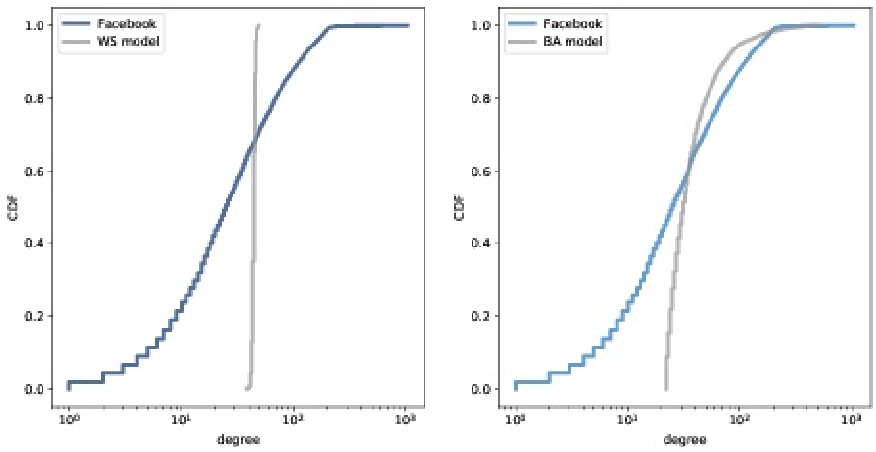


图 4.4：Facebook 数据集中的度的 CDF，以及 WS 模型（左边）和 BA 模型（右边），在双对数刻度上。

图 4.3 通过在双对数刻度上绘制概率质量函数（PMF）来表示度的分布。这就是 Barabási 和 Albert 呈现他们的结果的方式，这是幂律分布的文章中最常使用的表示。但是，这不是观察这样的数据的最好方法。

更好的选择是累积分布函数（CDF），它将 `x` 值映射为小于或等于 `x` 的值的比例。

给定一个 Pmf，计算累积概率的最简单方法是将 `x` 的概率加起来，包括 `x`：

```
def cumulative_prob(pmf, x):
    ps = [pmf[value] for value in pmf if value<=x]
    return sum(ps)
```

例如，给定数据集中的度的分布，`pmf_pf`，我们可以计算好友数小于等于 25 的比例：

```
>>> cumulative_prob(pmf_fb, 25)
0.506
```

结果接近 0.5，这意味着好友数的中位数约为 25。

因为 CDF 的噪音比 PMF 少，所以 CDF 更适合可视化。一旦你习惯了 CDF 的解释，它们可以提供比 PMF 更清晰的分布图像。

`thinkstats` 模块提供了一个称为 `Cdf` 的类，代表累积分布函数。我们可以用它来计算数据集中的度的 CDF。

```
from thinkstats2 import Cdf
cdf_fb = Cdf(degrees(fb), label='Facebook')
```

`thinkplot` 提供了一个函数，叫做 `Cdf` ，绘制累积分布函数。

```
thinkplot.Cdf(cdf_fb)
```

图 4.4 显示了 Facebook 数据集的度的 CDF ，以及 WS 模型（左边）和 BA 模型（右边）。 `x` 轴是对数刻度。

显然，WS 模型和数据集的 CDF 很大不同。BA 模式更好，但还不是很好，特别是对于较小数值。

在分布的尾部（值大于 100），BA 模型看起来与数据集匹配得很好，但是很难看出来。我们可以使用另一个数据视图，更清楚地观察数据：在对数坐标上绘制互补 CDF。

互补 CDF（CCDF）定义为：

```
CCDF(x) = 1 − CDF(x)
```

它很有用，因为如果 PMF 服从幂律，CCDF 也服从：

```
CCDF(x) =(x/x_m)^(-α)
```

其中 `x_m` 是最小可能值， `α` 是确定分布形状的参数。

对两边取对数：

```
logCCDF(x) = −α (logx − logx_m)
```

因此，如果分布服从幂定律，在双对数刻度上，我们预计 CCDF 是斜率为 `-α` 的直线。

图 4.5 以双对数刻度显示 Facebook 数据的度的 CCDF，以及 WS 模型（左边）和 BA 模型（右边）。

通过这种查看数据的方式，我们可以看到 BA 模型与分布的尾部（值大于 20）匹配得相当好。WS 模型没有。

4.8 解释性模型

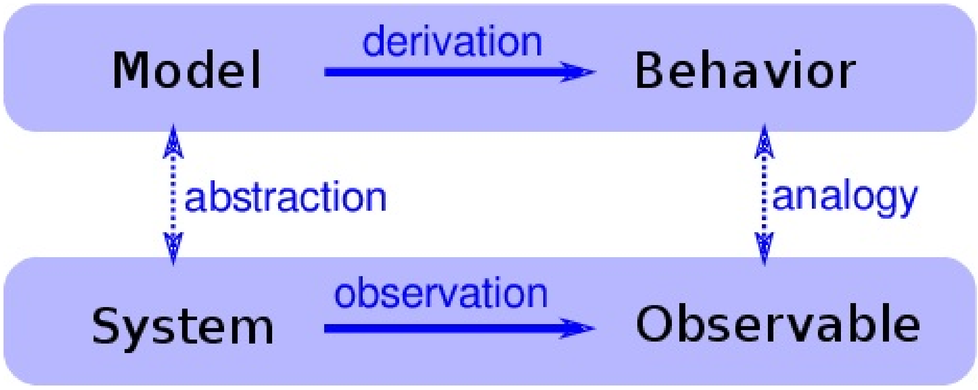


图 4.6：解释性模型的逻辑结构

我们以 Milgram 的小世界实验开始讨论网络，这表明社交网络中的路径长度是惊人的小；因此，有了“六度分离”。

当我们看到令人惊讶的事情时，自然会问“为什么”，但有时候我们不清楚我们正在寻找什么样的答案。一种答案是解释性模型（见图 4.6）。解释性模型的逻辑结构是：

1. 在一个系统 `S` 中，我们看到一些可观察的东西 `O`，值得解释。
2. 我们构建一个与系统类似的模型 `M`，也就是说，模型与系统之间的元素/组件/原理是对应的。
3. 通过模拟或数学推导，我们表明，该模型展现出类似于 `O` 的行为 `B`。
4. 我们得出这样的结论：`S` 表现 `O`，因为 `S` 类似于 `M`，`M` 表示 `B`，而 `B` 类似于 `O`。

其核心是类比论证，即如果两个事物在某些方面相似，那么它们在其他方面可能是相似的。

类比论证是有用的，解释模型可以令人满意，但是它们并不构成数学意义上的证明。

请记住，所有的模型都有所忽略，或者“抽象掉”我们认为不重要的细节。对于任何系统都有很多可能的模型，它们包括或忽略不同的特性。而且可能会出现不同的行为模式，`B`，`B'` 和 `B''`，这些模式与 `O` 不同。在这种情况下，哪个模型解释了 `O`？

小世界现象就是一个例子：Watts-Strogatz（WS）模型和 Barabási-Albert（BA）模型都展现出小世界行为的元素，但是它们提供了不同的解释：

- WS 模型表明，社交网络是“小”的，因为它们包括强连通的集群，和连接群集的“弱关系”（参见 http://en.wikipedia.org/wiki/Mark_Granovetter#The_strength_of_weak_ties）。
- BA 模型表明，社交网络很小，因为它们包括度较高的节点，作为中心，并且随着时间的推移，由于优先添加，中心会增长。

在科学的新兴领域，往往是这样，问题不是我们没有解释，而是它们太多。

4.9：练习

练习 1：

上一节中，我们讨论了小世界现象的两种解释，“弱关系”和“中心”。这些解释是否兼容？也就是说，他们能都对吗？你觉得哪一个解释更令人满意？为什么？

是否有可以收集的数据或可以执行的实验，它们可以提供有利于一种模型的证据？

竞争模型中的选择，是托马斯·库恩（Thomas Kuhn）的论文“客观性，价值判断和理论选择”（Objectivity, Value Judgment, and Theory Choice）的主题，你可以在 https://github.com/AllenDowney/ThinkComplexity2/blob/master/papers/kuhn.pdf 上阅读。

对于竞争模型中的选择，库恩提出了什么标准？这些标准是否会影响你对 WS 和 BA 模型的看法？你认为还有其他标准应该考虑吗？

练习 2：

NetworkX 提供了一个叫做`powerlaw_cluster_graph`的函数，实现了 Holme 和 Kim 算法，用于使用度的幂律分布和近似平均聚类，使图增长。阅读该函数的文档，看看是否可以使用它来生成一个图，节点数、度的均值和群聚系数与 Facebook 数据集相同。与实际分布相比较，模型中的度的分布如何？

练习 3：

来自 Barabási 和 Albert 论文的数据文件可从 http://www3.nd.edu/~networks/resources.htm 获得。他们的演员协作数据包含在名为`actor.dat.gz`的文件中。以下函数读取文件并构建图。

```python
import gzip

def read_actor_network(filename, n=None):
    G = nx.Graph()
    with gzip.open(filename) as f:
        for i, line in enumerate(f):
            nodes = [int(x) for x in line.split()]
            G.add_edges_from(thinkcomplexity.all_pairs(nodes))
            if n and i >= n:
                break
    return G
```

计算图中的演员数量和度的均值。以双对数刻度绘制度的 PMF。同时在对数-线性刻度上绘制度的 CDF，来观察分布的一般形状，并在双对数刻度上观察，尾部是否服从幂律。

注意：演员的网络不是连通的，因此你可能想要使用`nx.connected_component_subgraphs`查找节点的连通子集。

五、细胞自动机

> 原文：Chapter 5 Cellular Automatons
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

细胞自动机（CA）是一个世界的模型，带有非常简单的物理。“细胞”的意思是世界被分成一个大口袋，称为细胞。“自动机”是一台执行计算的机器 - 它可能是一台真机。 ，但更多时候，“机器”是数学抽象或计算机的模拟。

本章介绍了史蒂文沃尔夫勒姆（Steven Wolfram）在 20 世纪 80 年代进行的实验，表明一些细胞自动机展示出令人惊讶的复杂行为，包括执行任意计算的能力。

我讨论了这些结果的含义，在本章的最后，我提出了在 Python 中高效实现 CA 的方法。

本章的代码位于本书仓库的 `chap05.ipynb` 中。 使用代码的更多信息，请参见第？章。

5.1 简单的 CA

细胞自动机 [1] 由规则来管理，它决定系统如何即时演化。 时间分为离散的步骤，规则规定了，如何根据当前状态计算下一个时间步骤中的世界状态。

> automaton（自动机）的复数为 automatons 或 automata。

作为一个微不足道的例子，考虑带有单个细胞的细胞自动机（CA）。 细胞状态是用变量 `xi` 表示的整数，其中下标 `i` 表示 `xi` 是时间步骤 `i` 期间的系统状态。 作为初始条件， `x0 = 0` 。

现在我们需要一个规则。 我会任意选择 `xi = x[i-1] + 1` ，它表示在每个时间步骤之后，CA 的状态会增加 1。到目前为止，我们有一个简单的 CA ，执行简单的计算：它用于计数。

但是这个 CA 是不合规则的；可能的状态数通常是有限的。 为了使其成立，我将选择最小的感兴趣的状态数 2，和另一个简单的规则 `xi = (x[i-1] + 1) % 2` ，其中 `%` 是余数（或模）运算符。

这个 CA 的行为很简单：闪烁。 也就是说，在每个时间步之后，细胞的状态在 0 和 1 之间切换。

大多数 CA 是确定性的，这意味着规则没有任何随机元素；给定相同的初始状态，它们总是产生相同的结果。 也有不确定性的 CA，但我不会在这里涉及它们。

5.2 Wolfram 的实验

前一节中的 CA 只有一个细胞，所以我们可以将其视为零维，并且它不是很有趣。 在本章的其余部分中，我们将探索一维（1-D）CA，后者会变得非常有趣。

说 CA 有维度就是说细胞被安排在一个连续的空间中，这样它们中的一些可以看作“邻居”。 在一维中，有三种自然形式：

有限序列：

数量有限的细胞排成一排。 除第一个和最后一个之外的所有细胞都有两个邻居。

环：

数量有限的细胞排列成一个环。 所有细胞都有两个邻居。

无限序列：

数量无限的细胞排列成一排。

确定系统如何即时演化的规则，基于“邻域”的概念，即“邻域”，即决定给定细胞的下一个状态的一组细胞。

在二十世纪八十年代初，斯蒂芬沃尔夫勒姆发表了一系列论文，对一维 CA 进行了系统的研究。 他确定了四大类行为，每一类都比上一个更有趣。

Wolfram 的实验使用了三个细胞的邻域：细胞本身及其左右邻居。

在这些实验中，这些细胞有两个状态，分别表示为 0 和 1，所以规则可以通过一个表格进行汇总，它将邻域状态（状态的三元组）映射为中心细胞的下一个状态。 下表展示了一个示例：

prev	111	110	101	100	011	010	001	000
next	0	0	1	1	0	0	1	0

第一行显示邻居可能拥有的八个状态。第二行显示下一个时间步骤中的中心细胞的状态。 作为该表的简明编码，Wolfram 建议将第二行读作二进制数。 因为二进制 00110010 是十进制的 50，所以 Wolfram 称这个 CA 为“规则 50”。

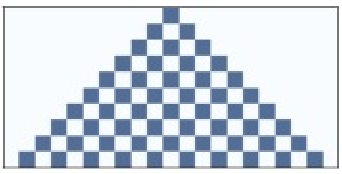


表 5.1：十个时间步骤之后的规则 50

上图展示了规则 50 在 10 个时间步骤之后的效果。 第一行展示第一个时间步骤内的系统状态;它起始于一个“开”细胞，其余都是“关”。 第二行展示下一个时间步骤中的系统状态，以此类推。

图中的三角形是这些 CA 的典型；这是领域形状的结果吗？ 在一个时间步骤中，每个细胞都会影响任一方向上的邻居的状态。 在下一个时间步骤中，该影响可以在每个方向上向一个细胞传播。 因此，过去的每个细胞都有一个“影响三角形”，包括所有可能受其影响的细胞。

5.3 CA 的分类

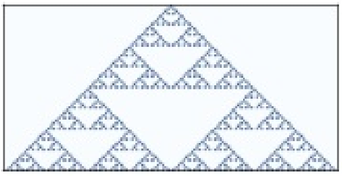


图 5.2：64 个步骤之后的规则 18

有多少种不同的 CA？

由于每个细胞都处于开或关的状态，我们可以用一位来指定细胞的状态。在三个细胞的邻域中，有 8 种可能的情况，因此规则表中有 8 个条目。由于每个条目都占一个位，我们可以使用 8 位指定一个表。使用 8 位，我们可以指定 256 个不同的规则。

Wolfram 的第一个 CA 实验就是测试所有 256 种可能性并尝试对它们进行分类。

在视觉上检查结果时，他提出 CA 的行为可以分为四类。第一类包含最简单（也是最不感兴趣）的 CA，即从几乎任何起始条件演变为相同统一图案的 CA。作为一个简单的例子，规则 0 总是在一个时间步后产生一个空的图案。

规则 50 是第二类的一个例子。它生成一个带有嵌套结构的简单图案；也就是说，该图案包含许多自身的较小版本。规则 18 使嵌套结构更加清晰；图？显示了 64 步后的样子。

这种模式类似于谢尔宾斯基三角形，你可以在 http://en.wikipedia.org/wiki/Sierpinski_triangle 上阅读。

某些二类 CA 生成的图案复杂而美观，但与第三和第四类相比，它们相对简单。

5.4 随机性

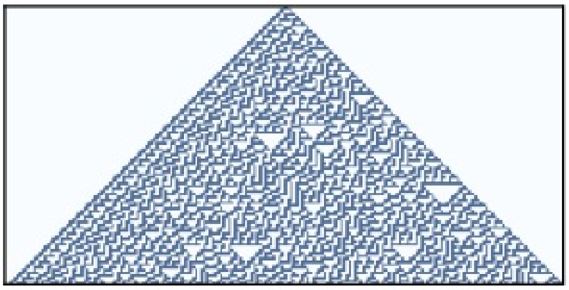


图 5.3：100 个步骤之后的规则 30

第三类包含产生随机性的 CA。规则 30 是一个例子；图？显示 100 个时间步后的样子。

左侧有一个明显的图案，右侧有各种大小的三角形，但中心看起来很随意。 事实上，如果你把中间列看做一个比特序列，就很难将其区分于真正的随机序列。 它通过了许多统计测试，人们用来测试比特序列是否随机。

产生看起来随机的数字的程序，称为伪随机数字生成器（PRNG）。 他们不被认为是真正的随机，因为：

- 它们中的许多产生规律性序列，可以通过统计来检测。 例如，C 标准库中的`rand`的原始实现，使用了线性同余生成器，生成器生成的序列具有易于检测的序列相关性。
- 任何使用有限状态（即存储）的 PRNG 最终都会重复。 生成器的特点之一就是这种重复周期。
- 底层过程基本上是确定性的，不同于一些物理过程，如放射性衰减和热噪声，被认为是基本随机的。

现代 PRNG 产生的序列，在统计上与随机值无法区分，并且它们以很长的周期实现，以至于在重复之前宇宙将崩溃。 这些发生器的存在，提出了一个问题，即质量好的伪随机序列与由“真正的”随机过程产生的序列之间，是否存在真正差异。 在“A New Kind of Science”中，沃尔夫勒姆认为没有（第 315-326 页）。

5.5 确定性

第三类 CA 的存在令人惊讶。 为了解释多么令人惊讶，让我从哲学确定性（决定论）开始（参见http://en.wikipedia.org/wiki/Determinism）。 许多哲学立场很难准确定义，因为它们有不同的风味。 我经常发现，使用从弱到强排列的陈述列表，来定义它们是有用的：

D1：

确定性模型可以对某些物理系统做出准确的预测。

D2：

许多物理系统可以用确定性过程建模，但有些系统本质上是随机的。

D3：

所有事件都是由先验事件造成的，但许多物理系统基本上是不可预测的。

D4：

所有事件都是由先验事件造成的，并且可以（至少原则上）预测。

我构建这个范围的目标是，让 D1 如此弱以至于几乎每个人都会接受它，D4 如此强以至于几乎没有人会接受它，并且有些人会接受中间的陈述。

作为对历史发展和科学发现的回应，世界舆论的质心沿着这个范围摆动。 在科学革命之前，许多人认为宇宙的运作基本上是不可预测的，或由超自然力量所控制。 在牛顿力学的胜利之后，一些乐观主义者开始相信像 D4 这样的东西；例如，皮埃尔-西蒙拉普拉斯（Pierre-Simon Laplace）在 1814 年写道：

> 我们可以把宇宙的现状看作过去的果和未来的因。 一个智能在某个特定的时刻，知道所有使自然运动的力量，以及构成自然的所有物品的所有位置，如果它也足够大，来提交这些数据用于分析，它会将宇宙最大的天体和最小的原子的运动汇总成一个公式； 对于这样的智能来说，没有什么是不确定的，未来就像过去一样会存在于它的眼前。

这种“智能”被称为“拉普拉斯的恶魔”。见 http://en.wikipedia.org/wiki/Laplace's_demon。 在这种情况下，“恶魔”这个词具有“精神”的意义，没有邪恶的含义。

19 世纪和 20 世纪的发现逐渐打破了拉普拉斯的希望。 热力学，放射性和量子力学对强式的决定论构成了连续的挑战。

在 20 世纪 60 年代，混沌理论表明，在某些确定性系统中，预测只能在短时间尺度上进行，并受初始条件测量精度的限制。

大多数这些系统，是空间连续（不是时间）和非线性的，所以它们行为的复杂性并不令人惊讶。 沃尔夫勒姆在简单的细胞自动机中展示的复杂行为更令人惊讶，并且令人不安，至少对于确定性的世界观来说。

到目前为止，我一直关注对确定性的科学挑战，但是最持久的反对意见是确定性与人类自由意志之间的冲突。 复杂性科学为这种明显的冲突提供了可能的解决方案; 我将在第？章中回到这个话题。

5.6 飞船

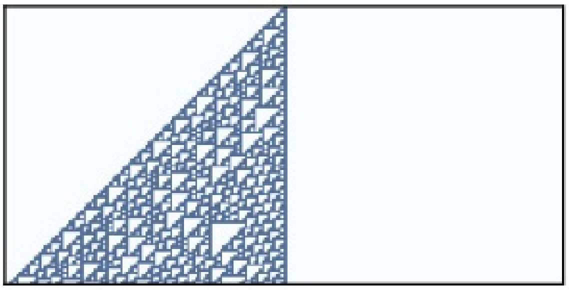


图 5.4：100 步之后的规则 110

第四类 CA 的行为更令人惊讶。 几个一维 CA，最着名的是规则 110，是图灵完备的，这意味着他们可以计算任何可计算的函数。 这个属性也称为普遍性，由 Matthew Cook 在 1998 年证明。请参阅 http://en.wikipedia.org/wiki/Rule_110。

图？展示了初始条件为单个细胞和 100 个时间步骤的规则 110 的样子。 在这个时间尺度上，没有发生什么特别的事情。 有一些有规律的模式，但也有一些难以表述的特征。

图？展示了更大的图像，它起始于一个随机的初始条件和 600 个时间步骤：

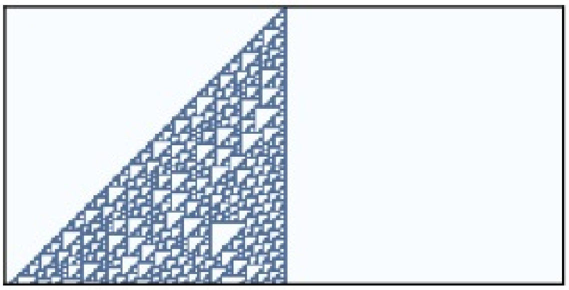


图 5.5：初始条件随机和 600 个时间步骤的规则 110

经过大约 100 个步骤后，背景变成了简单的重复模式，但背景中有一些持久性结构表现为干扰。 其中一些结构是稳定的，所以它们表现为垂直线条。 其他的在空间中平移，表现为不同斜率的对角线，取决于它们移动一列所需的时间步数。 这些结构被称为飞船。

飞船之间的碰撞产生不同的结果，取决于飞船的类型和它们碰撞时的阶段。 一些碰撞歼灭两艘船，其他船只保持不变；还有一些产生不同类型的一艘或多艘船只。

这些碰撞是 CA 规则 110 中的计算基础。 如果你将飞船视为通过电线传播的信号，并将碰撞视为计算 AND 和 OR 等逻辑运算的门，那么你可以看到 CA 执行计算的意义。

5.7 通用性

为了理解通用性，我们必须理解可计算性理论，它关于计算模型和计算的东西。

最通用的计算模型之一是图灵机，它是由艾伦图灵在 1936 年提出的一种抽象计算机。图灵机是一个一维 CA，两个方向上都是无限的，并增加了一个读写头。在任何时候，头部都位于一个细胞上。它可以读取该细胞的状态（通常只有两种状态），并可以将新值写入细胞中。

此外，该机器还有一个寄存器，用于记录机器的状态（有限数量的状态之一）和一张规则表。对于每个机器状态和细胞状态，表格规定一个操作。操作包括修改头部所在的细胞，并向左或向右移动一个细胞。

图灵机并不是计算机的实际设计，但它模拟了常见的计算机体系结构。对于在真实计算机上运行的给定程序，（至少原则上）可以构造一个执行等效计算的图灵机。

图灵机很有用，因为它可以刻画一组图灵机可以计算的函数，这就是图灵所做的事情。 这个集合中的函数被称为图灵可计算的。

说图灵机可以计算任何图灵可计算函数，是一个赘述：根据定义它是真的。 但图灵可计算性比这更有趣。

事实证明，任何人提出的每个合理的计算模型都是图灵完备的；也就是说，它可以计算与图灵机完全相同的一组函数。 其中一些模型，如 lamdba 演算，与图灵机非常不同，所以它们的等价性令人惊讶。

这种观察产生了丘奇-图灵理论，它基本上定义了可计算的含义。 这个“理论”是，图灵可计算性是可计算性的正确，或至少是自然定义，因为它描述了这种计算模型的多样化集合的威力。

CA 规则 110是另一种计算模型，其简单性非常出色。 它也是通用的，为丘奇-图灵理论提供了支持。

在“A New Kind of Science”中，沃尔夫勒姆阐述了这个理论的一个变种，他称之为“计算等价性原理”：

几乎所有不明显的简单过程，都可以看作是具有复杂性相同的计算。 更具体来说，计算等价性原理表明，在自然界中发现的系统可以执行达到最高（“通用”）级别的计算能力的计算，并且大多数系统实际上实现了这种最高级别的计算能力。 因此，大多数系统在计算上是等效的（参见 http://mathworld.wolfram.com/PrincipleofComputationalEquivalence.html）。

将这些定义应用于 CA，第一类和第二类“显然很简单”。 第三类可能不那么明显，但在某种程度上，完美的随机性就像完美的顺序一样简单；复杂性存在于中间。 所以 Wolfram 声称第四类行为在自然界中很常见，并且几乎所有表现它的系统在计算上都是等价的。

5.8 可证伪性

沃尔夫勒姆认为，他的原则比丘奇图灵理论更强大，因为它是关于自然界的，而不是抽象的计算模型。但是说自然过程“可以看作计算”，使我觉得像理论选择的陈述。而不仅仅是自然世界的假设。

此外，对于像“几乎”和“明显简单”这样的未定义术语的资格，他的假设可能是不可证伪的。可证伪性是科学哲学的一个观点，由卡尔波普尔（Karl Popper）提出，作为科学假说与伪科学之间的界限。如果一个假设是假的，并且有一个实验，至少在实用性领域，它能反驳这个假设，那么这个假设是可证伪的。

例如，地球上的所有生命都来自共同祖先的说法是可证伪的，因为它对现代物种（在其他东西中）的基因相似性做出了特定的预测。如果我们发现了一种新物种，它的 DNA 与我们的 DNA 几乎完全不同，那么这就反驳了共同血统理论（或者至少引起质疑）。

另一方面，所谓“神创论”，即所有物种都是由超自然力量创造出来的，是不可证实的，因为没有任何我们可以观察到的，与自然世界相矛盾的东西。任何实验的结果都可以归因于创作者的意志。

不可证伪的假设可能有吸引力，因为不可能反驳它们。如果你的目标是永远不会被证明是错误的，你应该尽可能选择不可证伪的假设。

但是，如果你的目标是对世界做出可靠的预测 - 而这至少是科学的目标之一 - 那么不可证伪的假设是无用的。问题是他们没有结果（如果他们有结果，他们将是可证伪的）。

例如，如果神创论是真实的，那我知道它有什么好处呢？它不会告诉我任何造物主的事情，除了他有一种“对甲虫的非常喜爱”（归因于 J. B. S. Haldane）。不同于共同血统理论，它通告许多科学和生物工程领域，理解这个世界或者为之行动是没有用的。

5.9 这是什么模型？

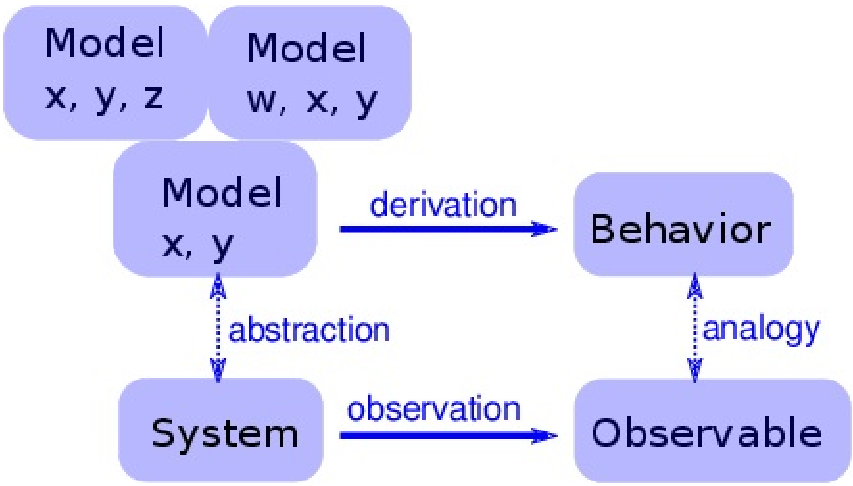


图 5.6：一个简单物理模型的逻辑结构

一些细胞自动机主要是数学工艺品。 它们很有趣，因为它们令人惊讶，或者有用，或者漂亮，或者因为它们提供了创建新式数学的工具（比如丘奇图灵定理）。

但是，它们是不是物理系统的模型还并不清楚。 如果他们是，他们是高度抽象的，也就是说他们并不很详细或现实。

例如，某些锥螺物种在它们的壳上产生图案，类似于由细胞自动机产生的图案（参见`en.wikipedia.org/wiki/Cone_snail`）。 所以假设 CA 是随着壳长大而在壳上产生图案的机制的模型，这是很自然的。 但是，至少在最初阶段，模型元素（所谓的细胞，邻居之间的通信，规则）如何对应成长的蜗牛（真实细胞，化学信号，蛋白质交互网络）的元素，还并不清楚。

对于传统的物理模型，现实是一种优点。如果模型的元素对应物理系统的元素，则模型和系统之间有明显的类比。总的来说，我们期望更现实的模型能够做出更好的预测，并提供更可信的解释。

当然，这只是一个事实。更详细的模型更难以处理，并且通常不太适合分析。在某些时候，模型变得如此复杂，以至于直接对系统进行实验更容易。

在另一个极端，简单的模型可以完全引人注目，因为它们很简单。

简单模型提供了与详细模型不同的解释。使用详细的模型，论述就像这样："我们对物理系统`S`感兴趣，所以我们构造了一个详细模型`M`，并且通过分析和模拟表明`M`表现出一种行为`B`，它与实际系统的观察`O`（定性或定量地）相似。那么为什么`O`会发生？因为`S`类似于`M`，而`B`类似于`O`，我们可以证明`M`导致`B`。"

使用简单的模型，我们不能说`S`与`M`相似，因为它不是。 相反，论述是这样的："有一组模型共享一组共同的特征。 任何具有这些特征的模型都表现出行为`B`。如果我们进行类似于`B`的观察`O`，解释它的一种方式是，这表明系统`S`具有足以产生`1`的一组特征。"

对于这种说法，增加更多的特征并没有帮助。 使模型更真实不会使模型更可靠；它只掩盖了导致`O`的基本特征，和`S`特有的附带特征之间的差异。

图？显示了这种模型的逻辑结构。 特征`x`和`y`足以产生行为。 增加更多细节，如特征`w`和`z`，可能会使模型更加逼真，但是这种现实并没有增加解释力。

5.10 CA 的实现

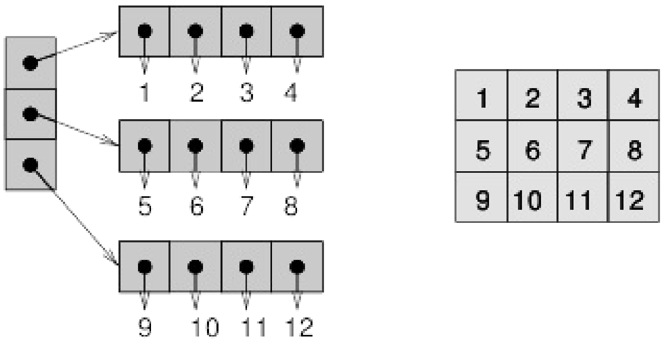


图 5.7：列表的列表（左）和 NumPy 数组（右）

为了生成本章中的图形，我编写了一个名为 CA 的 Python 类，它代表细胞自动机，以及用于绘制结果的类。在接下来的几节中，我会解释他们如何工作。

为了存储 CA 的状态，我使用了 NumPy 数组，这是一个多维数据结构，其元素类型都相同。它与嵌套列表类似，但通常更小更快。图？说明了原因。左侧的图展示了整数列表的列表；每个点表示一个引用，它占用 4-8 个字节。要访问其中的一个整数，你必须跟随两个引用。

右图显示了相同整数的数组。因为这些元素大小都相同，所以它们可以连续存储在内存中。这种安排节省了空间，因为它不使用引用，并且节省了时间，因为可以直接从下标计算元素的位置；没有必要跟随一系列的引用。

为了解释我的代码如何工作，我将以一个 CA 开始，它计算每个邻域中细胞的“奇偶性”。如果数字是偶数，则数字的奇偶性为 0；如果数字为奇数，则奇偶性为 1。

首先，我在第一行的中间，创建带有单个 1 的零数组。

```
>>> rows = 5
>>> cols = 11
>>> ca = np.zeros((rows, cols))
>>> ca[0, 5] = 1
print(ca)
[[ 0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

`plot_ca` 用图形展示了结果。

```
mport matplotlib.pyplot as plt

def plot_ca(ca, rows, cols):
    cmap = plt.get_cmap('Blues')
    plt.imshow(array, interpolation='none', cmap=cmap)
```

按照约定，我使用缩写名称 `plt` 引入了 `pyplot` 。 `imshow` 将数组视为“图像”并显示它。 使用颜色表 `'Blues'` ，将“开”细胞绘制为深蓝色，“关”细胞绘制为淡蓝色。

现在，为了计算下一个时间步中的 CA 状态，我们可以使用 `step` ：

```
def step(array, i):
    rows, cols = array.shape
    for j in range(1, cols):
        array[i, j] = sum(array[i-1, j-1:j+2]) % 2
```

参数 `ca` 是表示 CA 状态的 NumPy 数组。 `rows` 和 `col` 是数组的维数，而 `i` 是我们应该计算的时间步骤的索引。 我用 `i` 来表示数组的行，它们对应于时间，`j` 表示对应于空间的列。

在 `step` 内部，我们遍历第 `i` 行的元素。 每个元素是来自上一行的三个元素的总和，并对 2 取余。

5.11 互相关

上一节中的 `step` 函数很简单，但速度并不是很快。 一般来说，如果我们用 NumPy 操作替换循环，我们可以加速这样的操作，因为 Python 解释器中的 `for` 循环会产生大量开销。 在本节中，我将展示如何使用NumPy函数相关来加快步骤。

首先，我们可以使用数组乘法来代替切片运算符来选择邻域。 具体来说，我们将数组乘以一个窗口，其中我们想要选择的细胞为一，其余为零。

例如，以下窗口选择前三个元素：

```
>>> window = np.zeros(cols, dtype=np.int8)
>>> window[:3] = 1
>>> print(window)
[1 1 1 0 0 0 0 0 0 0 0]
```

如果我们乘以数组的最后一行，我们会得到前三个元素：

```
>>> print(array[4])
>>> print(window * array[4])
[0 1 0 0 0 1 0 0 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0]
```

现在我们可以使用 `sum` 和模运算符来计算下一行的第一个元素：

```
>>> sum(window * array[4]) % 2
1
```

如果我们将窗口向右移动，它会选择接下来的三个元素，以此类推。所以我们可以像这样重写 `step`：

```
def step2(array, i):
    rows, cols = array.shape
    window = np.zeros(cols)
    window[:3] = 1
    for j in range(1, cols):
        array[i, j] = sum(window * array[i-1]) % 2
        window = np.roll(window, 1)
```

`roll` 将窗口向右移动（它也把末尾的补在开头，但不影响这个函数）。

`step2` 产生 `step` 的相同结果。 它仍然不是非常快，但是它朝着正确的方向迈出了一步，因为我们刚刚执行的操作（乘以窗口，将结果相加，移动窗口并重复）用于各种应用。 它被称为互相关，而 NumPy 提供了一个称为 `correlate` 的函数来计算它。

我们可以用它来编写更快，更简单的步骤：

```
def step3(array, i):
    window = np.array([1, 1, 1])
    array[i] = np.correlate(array[i-1], window, mode='same') % 2
```

当我们使用 `np.correlate` 时，窗口不必与数组大小相同，因此使窗口更简单一些。

`mode` 参数决定结果的大小。 你可以阅读 NumPy 文档中的详细信息，但是当模式为 `'same'` 时，结果与输入大小相同。

5.12 CA 表

现在还差一步。 如果 CA 规则仅取决于邻居的总和，那么我们迄今为止的函数仍然有效，但大多数规则还取决于哪些邻居是开或者关的。 例如，100 和 001 可能会产生不同的结果。

我们可以使用一个包含元素 `[4,2,1]` 的窗口，使 `step` 更加通用，它将邻域解释为一个二进制数。 例如，邻域 100 产生 4；010 产生 2，001 产生 1。然后我们可以在规则表中查找这些结果。

以下是更一般的步骤：

```
def step4(array, i):
    window = np.array([4, 2, 1])
    corr = np.correlate(array[i-1], window, mode='same')
    array[i] = table[corr]
```

前两行几乎相同。 最后一行在 `table` 中查找 `corr` 的每个元素，并将结果赋给 `array[i]` 。

最后，这是计算表的函数：

```
def make_table(rule):
    rule = np.array([rule], dtype=np.uint8)
    table = np.unpackbits(rule)[::-1]
    return table
```

参数 `rule` 是一个 0 到 255 的整数。第一行将规则放入单个元素的数组中，以便我们可以使用 `unpackbits`，将规则编号转换为其二进制表示形式。 例如，以下是规则 150 的表：

```
>>> table = make_table(150)
>>> print(table)
[0 1 1 0 1 0 0 1]
```

在 `thinkcomplexity.py` 中，你将找到CA的定义，它封装了本节中的代码，以及两个绘制 CA 的类，`PyplotDrawer` 和 `EPSDrawer` 。

5.13 练习

练习 1

本章的代码位于本书仓库的 Jupyter 笔记本 `chap05.ipynb` 中。打开这个笔记本，阅读代码，然后运行单元格。你可以使用这个笔记本来做本章的练习。我的解决方案在 `chap05soln.ipynb` 中。

练习 2

这个练习要求你试验规则 110 以及它的一些飞船。

1. 阅读规则 110 的维基百科页面，其中描述了其背景图案和飞船：https://en.wikipedia.org/wiki/Rule_110。
2. 使用产生稳定背景图案的初始条件创建 CA 规则 110。

   请注意，CA 类提供了 `start_string`，它允许你使用 1 和 0 的字符串初始化数组的状态。
3. 通过在行的中心添加不同的图案来修改初始条件，并查看哪些产生了飞船。对于一些 `n` 的合理的值，你可能想列举所有可能的 `n` 位图案。对于每个飞船，你能找到平移的时间和速度吗？你能找到的最大的飞船是什么？
4. 当宇宙飞船相撞时会发生什么？

练习 3

这个练习的目标是实现一个图灵机。

1. 阅读 http://en.wikipedia.org/wiki/Turing_machine 来了解图灵机。

2. 编写一个名为 `Turing` 的类来实现图灵机。对于动作表，使用三个状态的 Busy Beaver 的规则。
3. 写一个名为 `TuringDrawer` 的类，该类生成一个图像，表示磁带状态以及磁头位置和状态。可能的外观的一个示例，请参阅 http://mathworld.wolfram.com/TuringMachine.html 。

练习 4

1. 本练习要求你执行并测试几个 PRNG。为了进行测试，你需要安装 DieHarder，你可以从 https://www.phy.duke.edu/~rgb/General/dieharder.php 下载 DieHarder，也可能为你的操作系统作为软件包提供。
2. 编写一个程序，实现 http://en.wikipedia.org/wiki/Linear_congruential_generator 中描述的线性同余生成器之一。使用 DieHarder 进行测试。
3. 阅读 Python `random` 模块的文档。它使用了什么 PRNG？测试它。
4. 使用几百个细胞实现 CA 规则 30，在合理的时间内以尽可能多的时间步骤运行它，然后将中心列输出为位序列。测试它。

练习 5

可证伪性是一个吸引人的和有用的想法，但在科学哲学家中，它并不普遍视为界限的解决方案，正如波普尔所声称的那样。

阅读 http://en.wikipedia.org/wiki/Fifiability 并回答以下问题。

1. 界限问题是什么？
2. 根据波普尔的说法，可证伪性是否解决了界限问题？
3. 给出两个理论示例，一个被认为是科学，另一个被认为是非科学，它们由可证伪性的标准成功地区分开。
4. 你能总结出科学哲学家和历史学家对波普尔的主张提出的，一个或多个反对意见吗？
5. 你是否有这样的感觉，即实践哲学家对波普尔的工作给予高度评价？

六、生命游戏

> 原文：Chapter 6 Game of Life
>
> 译者：飞龙
>
> 协议：CC BY-NC-SA 4.0
>
> 自豪地采用谷歌翻译

在本章中，我们考虑二维细胞自动机，特别是 John Conway 的生命游戏（GoL）。 像上一章中的一些 CA 一样，GoL 遵循简单的规则并产生令人惊讶的复杂行为。 就像沃尔夫勒姆的规则 110 一样，事实证明 GoL 是通用的；也就是说，至少在理论上它可以计算任何可计算的函数。

GoL 的复杂行为引发了科学哲学问题，特别是科学现实主义和工具主义的相关问题。 我讨论这些问题并提出扩展阅读的建议。

在本章的最后，我演示了如何在 Python 中高效实现 GoL。

本章的代码位于本书仓库的 `chap06.ipynb` 中。 使用代码的更多信息，请参见第？节。

6.1 Conway 的生命游戏

首先要研究的细胞自动机之一，也许是有史以来最受欢迎的一种，是称为“生命游戏”的二维 CA，简称 GoL。 它由 John H. Conway 开发并于 1970 年在《科学美国人》（Scientific American）的马丁加德纳（Martin Gardner）专栏中推广。 请参阅 http://en.wikipedia.org/wiki/Conway_Game_of_Life。

GoL 中的细胞排列在一个二维网格中，两个方向上都有限，或者首尾相接。 双向首尾相接的网格称为环面，因为它在地形上等同于多纳圈的表面。 见 http://en.wikipedia.org/wiki/Torus。

每个细胞有两个状态 - 生存和死亡 - 和八个邻居 - 东西南北和四个对角线。 这些邻居有时被称为“摩尔邻域”。

就像前面章节中的一维 CA 一样，生命游戏按照规则演变，这就像物理学的简单定律。

在 GoL 中，每个单元格的下一个状态取决于其当前状态和活动邻居的数量。 如果一个细胞是活的，如果它有两个或三个活动邻居就会生存，否则就会死亡。 如果一个细胞是死的，它将保持死亡，除非它恰好有三个邻居。

下表总结了这些规则：

当前状态	邻居数量	下一个状态
生存	2–3	生存
生存	0–1, 4–8	死亡
死亡	3	生存
死亡	0–2, 4–8	死亡

这种行为与真正的细胞生长大致类似：分离或过度拥挤的细胞死亡；它们在中等密度下蓬勃成长。

GoL 很受欢迎，因为：

有简单的初始条件产生令人惊讶的复杂行为。

有许多有趣的稳定图案：有些摆动（以不同的周期），有些像 Wolfram 的 CA 规则 110 中的飞船一样移动。 和规则 110 一样，GoL 是图灵完整的。

另一个产生兴趣的因素是康威的猜测 - 没有可以使活细胞数量无限增长的初始条件 - 以及他向任何可以证明或否定它的人提供的 50 美元赏金。

最后，计算机日益增加的可用性，使得自动化计算并以图形方式显示结果成为可能。

6.2 生命图案

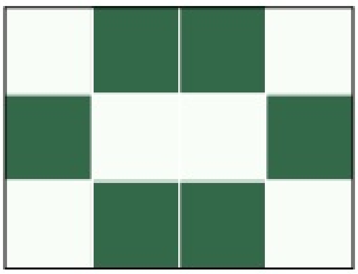


图 6.1：一个静态图案，叫做“蜂巢”（beehive）

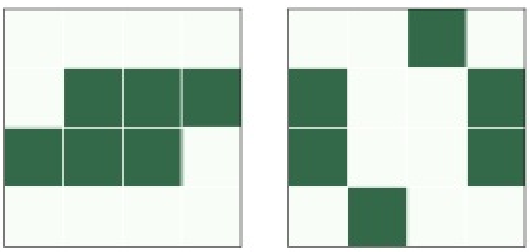


图 6.2：一个振荡图案，叫做“蟾蜍”（toad）

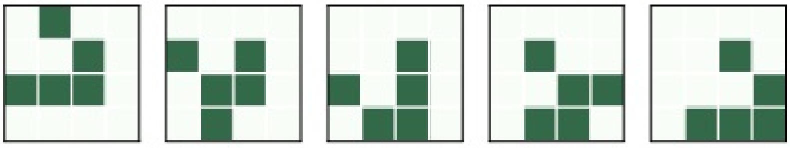


图 6.3：一个飞船，叫做“滑翔机”（glider）

如果从随机起始状态运行 GoL，可能会出现一些稳定图案。随着时间的推移，人们已经确定了这些图案并给了它们名字

例如，图？展示了一种称为“蜂巢”的稳定图案。蜂巢中的每个细胞都有两个或三个邻居，所以它们都能存活下来，蜂巢旁边的死细胞都没有三个邻居，所以没有新细胞诞生。

其他图案在“振荡”；也就是说，它们随着时间而改变，但最终返回到它们的起始状态（只要它们不与另一个图案冲突）。例如，图？展示了一种称为“蟾蜍”的图案，它是在两种状态之间交替的振荡图案。这个振荡图案的“周期”是二。

最后，一些图案振荡并返回到起始状态，但在空间中移动。因为这些图案似乎在移动，所以它们被称为“飞船”。

图？展示了一艘名为“滑翔机”的飞船。经过四段时间后，滑翔机回到起始位置，并向下和向右移动一个单位。

根据起始方向，滑翔机可以沿着四条对角线中的任何一条移动。还有其它的水平和垂直移动的飞船。

人们花费了大量时间来查找和命名这些图案。如果你搜索网页，你会发现很多收藏品。

6.3 Conwey 的推测

从最初的条件来看，GoL 迅速达到稳定状态，活细胞数量几乎不变（可能带有一些振荡）。

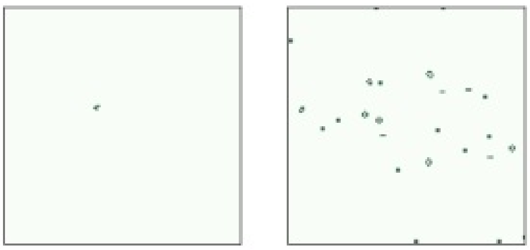


图 6.4：r-pentomino 的开始和最终状态

但是一些简单的开始条件，需要很长时间才能稳定下来，并产生令人惊讶的活细胞数量。 这些模式被称为“Methuselahs”，因为它们很长寿。

其中最简单的是 r-pentomino，它只有五个细胞，形状大致为字母“r”。 图？显示了 r-pentomino 的初始状态和 1103 步后的最终状态。

这种状态是“最终的”，因为所有剩余图案是稳定的，振荡的或滑翔机，它们永远不会与另一种图案相冲突。 r-pentomino 总共产生 6 个滑翔机，8 个积木（block），4 个闪光灯（blinker），4 个蜂巢，1 个小艇（boat），1 个轮船（ship）和 1 个面包（loaf）。

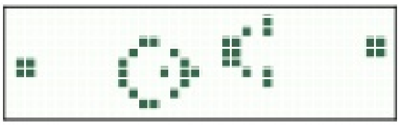


图 6.5：Gosper 的滑翔机枪，产生滑翔机流。

长寿图案的存在，使得康威怀疑是否存在从未稳定的初始图案。 他猜想没有，但他描述了两种证明他是错误的图案，“枪”（gun）和“蒸汽火车”（puffer train）。 枪是稳定的模式，定期产生飞船 - 随着飞船流从源位置移动，活细胞的数量无限增长。 蒸汽火车是一种将活细胞留在尾部的平移图案。

事实证明，这两种模式都存在。 由 Bill Gosper 领导的一个小组发现了第一个，它是现在称为 Gosper's Gun 的滑翔枪，如图所示。 Gosper 还发现了第一个蒸汽火车。

这两种类型都有很多图案，但它们很难设计或找到。 这不是巧合。 Conway 选择了 GoL 的规则，这样他的猜想就不会明显为真或假。 在二维 CA 的所有可能规则中，大多数产生简单的行为：大多数初始条件快速稳定或无限增长。 通过避免无趣的 CA，Conway 也避免了 Wolfram 的一类和二类行为，并且可能还有三类。

如果我们相信 Wolfram 的计算等价原则，我们预计 GoL 会属于第四类，而且是这样。 生命游戏在 1982 年被证明了图灵的完整性（1983年也是独立的）。 从那时起，几个人构建了 GoL 模式，实现了图灵机或另一台已知图灵完备的机器。

6.4 现实主义

GoL中的稳定模式很难不被注意，特别是那些移动的模式。 将它们视为持久的实体是很自然的事，但请记住，CA 是由细胞构成的；没有蟾蜍或面包这样的东西。 滑翔机和其他飞船甚至更不真实，因为随着时间的推移，它们甚至不由相同的细胞组成。 所以这些图案就像星座一

样。 我们这样看待他们，因为我们善于观察图案，或者因为我们有活跃的想象力，但他们不是真实的。

对嘛？

好吧，不是那样。 我们认为“真实”的许多实体，也是规模较小的实体的持久图案。 飓风只是气流的模式，但我们给了他们个人名称。 而人就像滑翔机，随着时间的推移不是由相同细胞组成的。 但即使你更换了你体内的每一个细胞，我们也认为你是同一个人。

这不是一个新观察 - 大约在 2500 年前，赫拉克利特（Heraclitus）指出你不能在同一条河流中两次 - 但是出现在生命游戏中的实体，是思考哲学现实主义的实用测试用例。

在哲学的背景下，现实主义是这样一种观点，即世界中的实体存在与人类的感知和概念无关。“感知”是指我们从感官中获得的信息，而“概念”是指我们形成的世界的心智模式。 例如，我们的视觉系统将一些东西感知为场景的二维投影，我们的大脑使用该图像构建场景中物体的三维模型。

科学实在论与科学理论和他们所假设的实体有关。 如果一个理论使用实体的属性和行为来表达，那么这个理论假设了一个实体。 例如，电磁学的理论用电场和磁场表示。 经济学的一些理论以供给，需求和市场力量来表达。 生物学的理论是用基因来表达的。

但这些实体是真实的吗？ 也就是说，它们存在于独立于我们和我们的理论的世界吗？

再次，我发现，在一系列强度中陈述哲学立场是有用的；这里有四个科学现实主义的陈述，强度逐渐增加：

SR1：

对于它们接近现实的程度，科学理论为真或假，但没有理论是完全正确的。 一些所假设的实体可能是真实的，但没有原则性的方式来说出哪些是真实的。

SR2：

随着科学的进步，我们的理论会变得更加逼近现实。 至少有一些所假定的实体是已知真实的。

SR3：

有些理论是完全正确的；其他近似真实。 真实理论所假设的实体，以及近似真实理论中的一些实体是真实的。

SR4：

如果一个理论正确地描述了现实，那么这个理论就是真的，否则就是假。真实理论所假设的实体是真实的；其他不是。

SR4 非常强，可能是站不住脚的；通过这样一个严格的标准，几乎所有当前的理论都被认为是错误的。 大多数现实主义者会接受 SR1 和 SR3 之间的东西。

6.5 工具主义

但 SR1 很弱以至于它接近工具主义，这是一种观点，我们不能说理论是眞是假，因为我们不知道理论是否符合现实。 理论是我们用于我们的目的的工具；在适用于其目的的程度上，理论是有用的，或者不是。

要看看你是否对工具主义感到满意，请考虑以下陈述：

“生命游戏中的实体并不是眞实的；他们只是人们赋予可爱的名字的细胞图案。”

“飓风只是一种气流模式，但它是一种有用的描述，因为它可以让我们进行有关天气的预测和沟通。”

“像本我和超我这样的弗洛伊德实体并不是眞实的，但它们是思考和交流心理学的有用工具（或者至少有些人是这么认为的）。”

“电磁场是我们最好的电磁理论中的假设实体，但它们并不眞实。 我们可以构建其他理论，而不用场的假设，这也是一样有用的。”

“我们认为，世界上的许多物体都是像星座一样的任意集合。 例如，蘑菇只是眞菌的子实体，其中大部分是在地下生长的，几乎不连续的细胞网络。 我们由于实际原因专注于蘑菇，如可见性和可爱。”

“有些物体边界清晰，但很多都是模糊的。 例如，哪些分子是你身体的一部分：你的肺里的空气？ 你的胃里的食物？ 你血液中的营养物质？ 细胞中的营养物质？ 细胞中的水？ 细胞的结构部分？ 头发？ 死皮？ 污垢？ 你的皮肤上的细菌？ 你的肠道细菌？线粒体？ 当你称量自己时，你包含了多少这些分子？ 根据离散对象构想世界是有用的，但我们确定的实体并不是眞实的。”

对于每一个你同意的陈述，给自己一分。 如果你的分数超过 4 分，你可能会成为一名工具主义者！

如果你比其他人更喜欢这些陈述，那么问问你自己为什么。 这些情景中的哪些差异会影响你的反应？ 你能否在他们之间做出原则性区分？

工具主义的更多信息，请参阅 http://en.wikipedia.org/wiki/Instrumentalism。

6.6 实现

本章最后的练习要求你尝试和修改生命游戏，并实现其他二维细胞自动机。 本节介绍 GoL 的实现，你可以将其用作实验的起始位置。

为了表示细胞的状态，我使用类型为 `uint8` 的 NumPy 数组，它是一个 8 位无符号整数。 例如，下面这行创建一个 10 乘 10 的数组，并用 0 和 1 的随机值进行初始化。

```
a = np.random.randint(2, size=(10, 10)).astype(np.uint8)
```

我们可以用几种方法计算 GoL 规则。 最简单的方法是使用 `for` 循环遍历数组的行和列：

```
b = np.zeros_like(a)
rows, cols = a.shape
for i in range(1, rows-1):
    for j in range(1, cols-1):
        state = a[i, j]
        neighbors = a[i-1:i+2, j-1:j+2]
        k = np.sum(neighbors) - state
        if state:
            if k==2 or k==3:
                b[i, j] = 1
        else:
            if k == 3:
                b[i, j] = 1
```

最初，`b` 是一个与 `a` 大小相同的零数组。 每次循环中，状态是中心细胞的条件，邻居是 `3x3` 的邻域。 `k` 是活动邻居的数量（不包括中心细胞）。 嵌套的 `if` 语句评估 GoL 规则并相应地激活 `b` 中的细胞。

这个实现是规则的直接翻译，但它是冗长而缓慢的。 我们可以使用互相关做得更好，正如我们在第？节中看到的那样。 在那里，我们使用 `np.correlate` 来计算一维相关。 现在，为了计算二维相关，我们将使用 `scipy.signal` 中的 `correlate2d`，它是一个 SciPy 模块，提供信号处理的相关函数：

```
from scipy.signal import correlate2d

kernel = np.array([[1, 1, 1],
                   [1, 0, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
```

在一维相关的背景下，我们称之为“窗口”的内容，在二维相关的背景下被称为“核”，但其想法是相同的： `correlate2d` 将核和数组相乘来选择一个邻域，然后将结果加起来。 这会核选择中心细胞周围的 8 个邻居。

`correlate2d` 将核应用于数组中的每个位置。 使用 `mode ='same'` 时，结果与 `a` 的大小相同。

现在我们可以使用逻辑运算符来计算规则：

```
b = (c==3) | (c==2) & a
b = b.astype(np.uint8)
```

第一行计算了一个布尔数组，其中应该有活细胞的地方为 `True`，其他地方为 `False`。 然后，`astype` 将布尔数组转换为整数数组。

这个版本更快，也许够好，但是我们可以通过修改核来简化它：

```
kernel = np.array([[1, 1, 1],
                   [1,10, 1],
                   [1, 1, 1]])

c = correlate2d(a, kernel, mode='same')
b = (c==3) | (c==12) | (c==13)
b = b.astype(np.uint8)
```

这个版本核的包含中心单元并赋予其权重 10。如果中心单元为 0，则结果介于 0 和 8 之间;如果中心单元为 1，则结果在 10 到 18 之间。使用这个核，我们可以简化逻辑运算，只选择值为 3，12 和 13 的细胞。

这看起来可能不是什么大的改进，但它允许进一步简化：使用这个核，我们可以使用一个表来查找细胞的值，就像我们在第？节中所做的那样。

```
table = np.zeros(20, dtype=np.uint8)
table[[3, 12, 13]] = 1
c = correlate2d(a, kernel, mode='same')
b = table[c]
```

除了位置 3，12 和 13 以外，表格中的任何位置都为零。当我们使用 `c` 作为表格中的索引时，NumPy 执行逐元素查找；也就是说，它从 `c` 中获取每个值，在表中查找它并将结果放入 `b` 中。

这个版本比其他版本更快更简洁， 唯一的缺点是需要更多的解释。

包含在本书仓库中的 `Life.py` 提供了一个封装规则实现的 `Life` 类。 如果你执行 `Life.py`，你应该看到一个“蒸汽火车”的动画，这是一种飞船，在其尾部留下一串碎屑。

6.7 练习

练习 1

本章的代码位于本书仓库的 Jupyter 笔记本 `chap06.ipynb` 中。 打开这个笔记本，阅读代码，然后运行单元格。 你可以使用这个笔记本来练习本章的练习。 我的解决方案在 `chap06soln.ipynb` 中。

练习 2

以随机状态启动 GoL 并运行它直至稳定。 你可以识别哪些稳定的图案？

练习 3

许多命名图案都以便携式文件格式提供。 修改 `Life.py` 来解析其中一种格式并初始化网格。

练习 4

一种最长寿的小型图案是“兔子”，它以 9 个活动细胞开始，需要 17 331 个步骤来稳定。 你可以在 http://www.conwaylife.com/wiki/Rabbits 获取各种格式的初始状态。 加载此状态并运行它。

练习 5

在我的实现中，`Life` 类基于一个名为 `Cell2D` 的父类，`LifeViewer` 基于 `Cell2DViewer`。 你可以使用这些基类来实现其他二维细胞自动机。

例如，GoL 的一个变体叫做“Highlife”，与 GoL 规则相同，另外还有一条规则：有 6 个邻居的死亡细胞会变活。

编写一个名为 `Highlife` 的类，该类继承自 `Cell2D` 并实现这个版本的规则。 另外编写一个名为 `HighlifeViewer` 的类，该类继承自 `Cell2DViewer` 并尝试以不同的方式来展示结果。 作为一个简单的例子，使用不同的颜色表。

`Highlife` 中更有趣的图案之一是复制器（replicator）。 使用 `add_cells` 和复制器初始化 `Highlife` 并查看它做了什么。

练习 6

如果将图灵机扩展到两个维度，或者将读写头添加到二维 CA，则结果是称为 Turmite 的细胞自动机。由于读写头移动的方式，它以白蚁（termite）命名，但拼写错误是对 Alan Turing 的敬意。

最着名的 Turmite 是 1986 年由 Chris Langton 发现的兰顿的蚂蚁（Langton's Ant）。请见 http://en.wikipedia.org/wiki/Langton_ant。

蚂蚁（ant）是一个具有四种状态的读写头，你可以将其视为面向东、西、南或北。细胞有两种状态，黑色和白色。

规则很简单。在每个时间步骤中，蚂蚁检查它所在单元格的颜色。如果是黑色，蚂蚁转向右转，将细胞变成白色，并向前移动一个格子。如果细胞是白色的，蚂蚁会向左转，将细胞变成黑色，然后向前移动。

给定一个简单的世界，一组简单的规则，并且只有一个可移动的部分，你可能会期望看到简单的行为 - 但你现在应该更清楚。从所有的白色细胞开始，在进入周期为 104 步的循环之前，兰顿的蚂蚁以看似随机的方式移动超过 10000 步。每个循环后，蚂蚁都会沿对角线平移，因此会留下一条称为“高速路”的踪迹。

编写兰顿的蚂蚁的实现。

七、物理建模

> 原文：Chapter 7 Physical modeling
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

到目前为止，我们所看到的细胞自动机不是物理模型；也就是说，他们不打算描述现实世界中的系统。 但是一些 CA 用作物理模型。

在本章中，我们考虑一个 CA，它模拟扩散（散开）并相互反应的化学物质，这是 Alan Turing 提出的过程，用于解释一些动物模式如何发展。

我们将试验一种 CA，它模拟通过多孔材料的渗透液体，例如通过咖啡渣的水。 这个模型是展示相变行为和分形几何的几个模型中的第一个，我将解释这两者的含义。

本章的代码位于本书仓库的 `chap07.ipynb` 中。 使用代码的更多信息，请参见第？节。

7.1 扩散

1952 年，艾伦图灵发表了一篇名为“形态发生的化学基础”的论文，该论文描述了涉及两种化学物质的系统行为，它们在空间中扩散并相互反应。 他表明，这些系统根据扩散和反应速率产生了广泛的模式，并推测像这样的系统可能是生物生长过程中的重要机制，特别是动物着色模式的发展。

图灵模型基于微分方程，但也可以使用细胞自动机来实现。

但在我们开始使用图灵模型之前，我们先从简单的事情开始：只有一种化学物质的扩散系统。 我们将使用 2-D CA，其中每个细胞的状态是连续的数量（通常在 0 和 1 之间），表示化学物质的浓度。

我们将通过比较每个细胞与其邻居的均值，来建模扩散过程。 如果中心细胞的浓度超过领域均值，则化学物质从中心流向邻居。 如果中心细胞的浓度较低，则化学物质以另一种方式流动。

以下核计算每个细胞与其邻居均值之间的差异：

```
kernel = np.array([[0, 1, 0],
                   [1,-4, 1],
                   [0, 1, 0]])
```

使用 `np.correlate2d` ，我们可以将这个核应用于数组中的每个细胞：

```
c = correlate2d(array, kernel, mode='same')
```

我们将使用一个扩散常数 `r` ，它关联了浓度差与流速：

```
array += r * c
```

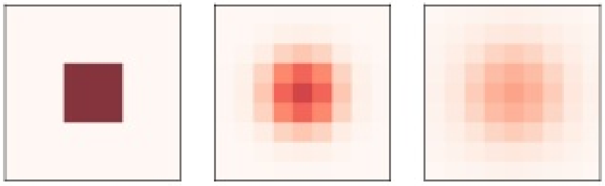


图 7.1：0，5 和 10 步后的简单扩散模型

图？显示 CA 的结果，其中 `n=9, r=0.1` ，除了中间的“岛”外，初始浓度为 0。 该图显示了 CA 的启动状态，以及 5 步和 10 步之后的状态。 化学物质从中心向外扩散，直到各处浓度相同。

7.2 反应扩散

现在我们添加第二种化学物。 我将定义一个新对象 `ReactionDiffusion` ，它包含两个数组，每个化学物对应一个：

```
class ReactionDiffusion(Cell2D):

    def __init__(self, n, m, params):
        self.params = params
        self.array = np.ones((n, m), dtype=float)
        self.array2 = np.zeros((n, m), dtype=float)
        island(self.array2, val=0.1, noise=0.1)
```

`n` 和 `m` 是数组中的行数和列数。 `params` 是参数元组，下面我会解释它。

数组代表第一种化学物质 `A` 的浓度，它最初是无处不在的。

`array2` 表示 `B` 的浓度，除了中间的一个岛屿，它初始为零，并且由 `island` 初始化：

```
def island(a, val, noise):
    n, m = a.shape
    r = min(n, m) // 20
    a[n//2-r:n//2+r, m//2-r:m//2+r] = val
    a += noise * np.random.random((n, m))
```

岛的半径 `r` 是 `n` 或 `m` 的二十分之一，以较小者为准。岛的高度是 `val`，在这个例子中是 `0.1`。此外，随机均匀噪声（值为 0 到 `noise`）添加到整个数组。

这里是更新数组的 `step` 函数：

```
def step(self):
    """Executes one time step."""
    A = self.array
    B = self.array2
    ra, rb, f, k = self.params

    cA = correlate2d(A, self.kernel, **self.options)
    cB = correlate2d(B, self.kernel, **self.options)

    reaction = A * B**2
    self.array += ra * cA - reaction + f * (1-A)
    self.array2 += rb * cB + reaction - (f+k) * B
```

参数是

`ra`：

`A` 的扩散速率（类似于前一节中的 `r`）。

`rb`：

`B` 的扩散速率。在该模型的大多数版本中，`rb` 约为 `ra` 的一半。

`f`：

进给速率，控制着 `A` 添加到系统的速度。

`k`：

移除速率，控制 `B` 从系统中移除的速度。

现在让我们仔细看看更新语句：

```
reaction = A * B**2
self.array += ra * cA - reaction + f * (1-A)
self.array2 += rb * cB + reaction - (f+k) * B
```

数组 `cA` 和 `cB` 是将扩散核应用于 `A` 和 `B` 的结果。乘以 `ra` 和 `rb` 得出进入或离开每个细胞的扩散速率。

表达式`A * B ** 2`表示`A`和`B`相互反应的比率。 假设反应消耗`A`并产生`B`，我们在第一个方程中减去这个项并在第二个方程中加上它。

表达式`f * (1-A)`决定`A`加入系统的速率。 当`A`接近 0 时，最大进给速率为`f`。 当`A`接近 1 时，进给速率下降到零。

最后，表达式`(f+k) * B`决定`B`从系统中移除的速率。 当`B`接近 0 时，该比率变为零。

只要速率参数不太高，`A`和`B`的值通常保持在 0 和 1 之间。

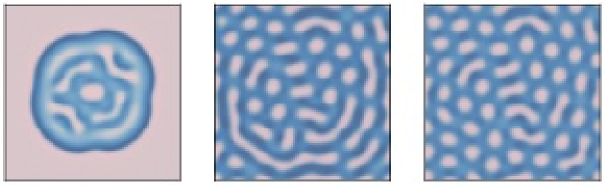


图 7.2：1000，2000 和 4000 步之后的反应扩散模型，参数为`f=0.035`和`k=0.057`

使用不同的参数，该模型可以产生类似于各种动物身上的条纹和斑点的图案。 在某些情况下，相似性是惊人的，特别是当进给和移除参数在空间上变化时。

对于本节中的所有模拟，`ra = 0.5`，`rb = 0.25`。

图？显示了`f=0.035`和`k=0.057`的结果，`B`的浓度以较暗的颜色显示。 有了这些参数，系统就向稳定状态演化，在`B`的黑色背景上有`A`的光点。

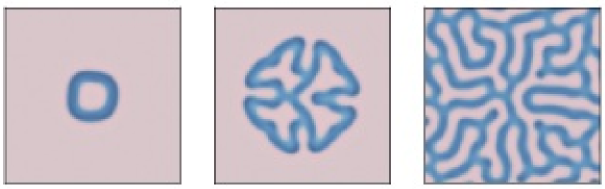


图 7.3：1000，2000 和 4000 步之后的反应扩散模型，参数为`f=0.055`和`k=0.062`

图？显示了`f = 0.055`和`k = 0.062`的结果，在`A`的背景上产生了珊瑚样的`B`。

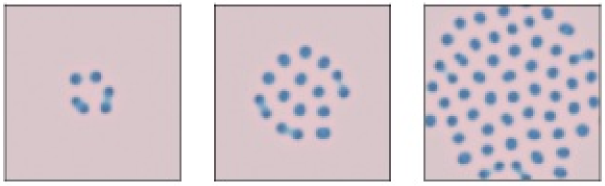


图 7.4：1000，2000 和 4000 步之后的反应扩散模型，参数为 `f=0.039` 和 `k=0.065`

图？显示了 `f = 0.039` 和 `k = 0.065` 的结果。 在类似于有丝分裂的过程中，这些参数产生的 `B` 点生长和分裂，最后形成稳定的等距点图案。

1952 年以来，观察和实验为图灵猜想提供了一些支持。 目前为止，看起来许多动物图案实际上由某种反应扩散过程形成，但尚未证实。

7.3 渗流

渗流是流体流过半多孔材料的过程。 实例包括岩层中的油，纸中的水和微孔中的氢气。 渗流模型也用于研究不是严格渗滤的系统，包括流行病和电阻网络。 请见 http://en.wikipedia.org/wiki/Percolation_theory。

渗流模型常常用随机图来表示，就像我们在第？章中看到的那样，但它们也可以用细胞自动机表示。 在接下来的几节中，我们将探索模拟渗流的 2-D CA。

在这个模型中：

- 最初，每个细胞是概率为 `p` 的“多孔”或者“无孔”，并且除了顶部那行是“湿的”之外，所有单元都是“干的”。
- 在每个时间步骤中，如果多孔细胞至少有一个湿的邻居，它会变湿。 非多孔细胞保持干燥。
- 模拟运行直至达到不再有细胞改变状态的“固定点”。

如果存在从顶部到底部的湿细胞路径，我们说 CA 具有“渗流簇”。

渗流的一个主要问题是，找到渗流簇的概率以及它如何依赖于 `p`。 这个问题可能会让你想起第？节，其中我们计算了随机 ER 图连接的概率。 我们会看到这两个模型之间的几个关系。

我定义了一个新类来表示渗流模型：

```python
class Percolation(Cell2D):

    def __init__(self, n, m, p):
        self.p = p
        self.array = np.random.choice([0, 1], (n, m), p=[1-p, p])
        self.array[0] = 5
```

`n` 和 `m` 是 CA 中的行数和列数。 `p` 是细胞为多孔的概率。

CA 的状态存储在数组中，该数组使用 `np.random.choice` 初始化，以概率 `p` 选择 1（多孔），以概率 `1-p` 选择 0（无孔）。 顶部那行的状态设置为 5，表示一个湿细胞。

在每个时间步骤中，我们使用 4 细胞邻域（不包括对角线）来检查任何多孔细胞是否拥有湿的邻居。 这是核：

```python
kernel = np.array([[0, 1, 0],
                   [1, 0, 1],
                   [0, 1, 0]])
```

这里是 `step` 函数：

`correlate2d` 将邻居的状态相加，如果至少有一个邻居是湿的，那么至少大于 5。 最后一行寻找多孔的细胞， `a == 1` ，并且至少有一个湿邻居， `c >= 5` ，并将它们的状态设置为 5，这代表湿的。



图 7.5：渗流模型的前三个步骤，其中 `n=10` 和 `p=0.5`

图？显示了 `n = 10` 和 `p = 0.5` 的渗流模型的前几个步骤。 非多孔细胞为白色，多孔细胞为浅色，湿细胞为深色。

7.4 相变

现在让我们测试 CA 是否包含渗流簇。

```python
def test_perc(perc):
    num_wet = perc.num_wet()

    num_steps = 0
    while True:
        perc.step()
        num_steps += 1

        if perc.bottom_row_wet():
            return True, num_steps

        new_num_wet = perc.num_wet()
        if new_num_wet == num_wet:
            return False, num_steps

        num_wet = new_num_wet
```

`test_perc` 接受 `Percolation` 对象作为参数。 每次循环中，它都会使 CA 前进一个时间步骤。 它检查底部那行，看看有没有湿的细胞；如果有，它返回 `True` ，表示存在渗透簇，以及 `num_steps` ，它是到达底部所需的时间步数。

在每个时间步骤中，它还计算湿细胞的数量并检查自上一步以来数量是否增加。 如果没有，我们已经到达了固定点，而没有找到一个渗流簇，所以我们返回 `False` 。

为了估计渗流簇的概率，我们生成许多随机初始状态并测试它们：

```python
def estimate_prob_percolating(p=0.5, n=100, iters=100):
    count = 0
    for i in range(iters):
        perc = Percolation(n, p=p)
        flag, _ = test_perc(perc)
        if flag:
            count += 1

    return count / iters
```

`estimate_prob_percolating` 使用给定的 `p` 和 `n` 值生成 100 个 CA，并调用 `test_perc` 来查看其中有多少个具有渗流簇。 返回值是拥有的 CA 的比例。

当 `p = 0.55` 时，渗滤簇的概率接近于 0。 `p = 0.60` 时，它约为 70%，而在 `p = 0.65` 时，它接近于 1。这种快速转变表明 `p` 的临界值接近 0.6。

我们可以更精确地使用随机游走来估计临界值。 从 `p` 的初始值开始，我们构造一个 `Percolation` 对象并检查它是否具有渗透簇。 如果是这样， `p` 可能太高，所以我们减少它。 如果不是， `p` 可能太低，所以我们增加它。

这里是代码：

```
def find_critical(p=0.6, n=100, iters=100):
    ps = [p]
    for i in range(iters):
        perc = Percolation(n=n, p=p)
        flag, _ = test_perc(perc)
        if flag:
            p -= 0.005
        else:
            p += 0.005
        ps.append(p)
    return ps
```

`find_critical` 以 `p` 的给定值开始并上下调整，返回值的列表。 当 `n = 100` 时，`ps` 的平均值约为 0.59，对于从 50 到 400 的 `n` 值，这个临界值似乎是一样的。

临界值附近的行为的快速变化称为相变，类似于物理系统中的相变，例如水在冰点处从液体变为固体的方式。

在处于或接近临界点时，各种各样的系统展示了一组共同的行为和特征。这些行为被统称为临界现象。 在下一节中，我们将探究其中的一个：分形几何。

7.5 分形

为了理解分形，我们必须从维度开始。

对于简单的几何对象，维度根据缩放行为而定义。 例如，如果正方形的边长为 `l`，则其面积为 `l ** 2`。 指数 2 表示正方形是二维的。 同样，如果立方体的边长为 `l`，则其体积为 `l ** 3`，这表示立方体是三维的。

更一般来说，我们可以通过测量一个对象的“尺寸”（通过一些定义），将对象的维度估计为线性度量的函数。

例如，我将通过测量一维细胞自动机的面积（“开”细胞的总数），将它的维度估计为行数的函数。

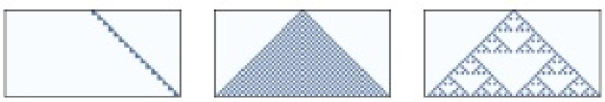


图 7.6：32 个时间步之后，规则为 20，50 和 18 的一维 CA。

图？展示了三个一维 CA，就像我们在第？节中看到的那样。 规则 20（左）产生一组看似线性的细胞，所以我们预计它是一维的。 规则 50（中）产生类似于三角形的东西，所以我们预计它是二维的。 规则 18（右）也产生类似三角形的东西，但密度不均匀，所以其缩放行为并不明显。

我将用以下函数来估计这些 CA 的维度，该函数计算每个时间步之后的细胞数。 它返回一个元组列表，其中每个元组包含 `i` 和 `i ** 2`，用于比较，以及细胞总数。

```
def count_cells(rule, n=500):
    ca = Cell1D(rule, n)
    ca.start_single()

    res = []
    for i in range(1, n):
        cells = np.sum(ca.array)
        res.append((i, i**2, cells))
        ca.step()

    return res
```

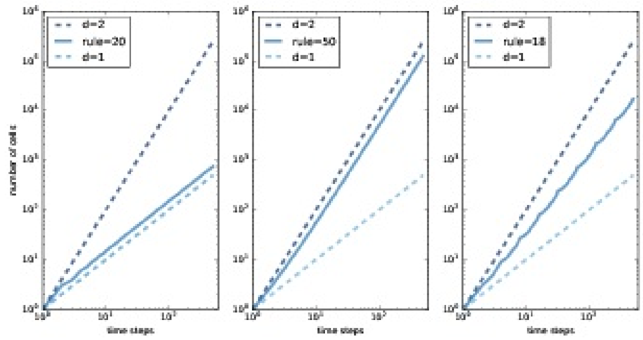


图 7.7：规则 20，50 和 18 的“开”细胞的数量与时间步数。

图？展示以双对数刻度绘制的结果。

在每幅图中，顶部虚线表示 `y = i ** 2` 。 两边取对数，我们得到 `logy = 2logi` 。 由于该数字在双对数刻度上，因此直线的斜率为2。

同样，底部的虚线表示 `y = i` 。 在双对数刻度上，直线的斜率为 1。

规则 20（左）每两个时间步骤产生三个细胞，所以 `i` 步后的细胞总数为 `y = 1.5 i` 。 两边取对数，我们得到 `logy = log1.5 + logi` ，所以在双对数刻度上，我们期待一条斜率为 1 的线。实际上，线的估计的斜率为 1.01。

规则 50（中）在第 `i` 个时间步骤中产生 `i + 1` 个新细胞，因此 `i` 步之后的细胞总数为 `y = i ** 2 + i` 。 如果我们忽略第二项并取两边的对数，我们有 `logy ~ 2 logi` ，所以当 `i` 变大时，我们预计看到一条斜率为 2 的线。事实上，估计的斜率为 1.97。

最后，对于规则 18（右），估计的斜率大约是 1.57，这显然不是 1，2 或任何其他整数。 这表明规则 18 生成的图案具有“分数维度”；也就是说，它是一个分形。

7.6 分形和渗流模型

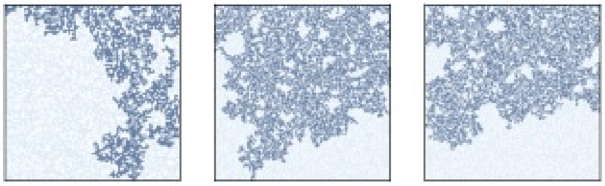


图 7.8： `p=0.6` 和 `n=100, 200, 300` 的渗流模型

现在让我们回到渗透模型。 图？展示了 `p = 0.6` 和 `n = 100, 200, 300` 的渗流模型中的湿细胞簇。非正式来说，它们类似于在自然界和数学模型中看到的分形模式。

为了估计它们的分形维度，我们可以运行一系列尺寸的 CA，计算每个渗流簇中湿细胞的数量，然后看看随着我们增加 CA 的大小，细胞计数的规模如何增长。

以下循环运行了模拟：

```
for size in sizes:
    perc = Percolation(size, p=p)
    flag, _ = test_perc(perc)
    if flag:
        num_filled = perc.num_wet() - size
        res.append((size, size**2, num_filled))
```

结果是元组列表，其中每个元组包含 `size` 和 `size ** 2` ，用于比较，以及渗流簇中的细胞数（不包括顶行中的初始湿细胞）。

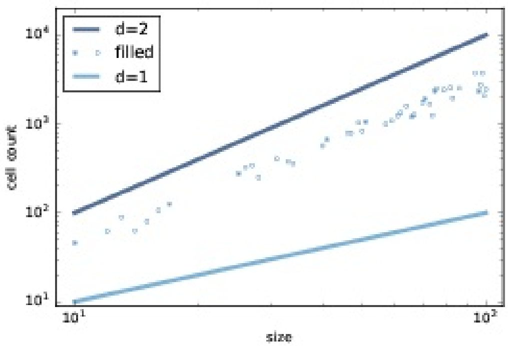


图 7.9：渗流簇中的细胞数量与 CA 大小

图？展示了 10 到 100 范围内的结果。点展示了每个渗流簇中的细胞数。 拟合这些点的线的斜率大约为 1.85，这表明当 `p` 接近临界值时，渗滤簇实际上是分形的。

当`p`大于临界值时，几乎每个多孔细胞都被填充，因此湿单元的数量仅为`p * size ** 2`，它的维度为 2。

当`p`远小于临界值时，湿细胞的数量与 CA 的线性大小成比例，因此它的维度为 1。

7.7 练习

练习 1

在第？节中，我们发现 CA 规则 18 产生了一个分形。 你能找到其他产生分形的一维 CA 吗？

注意：`Cell1D.py`中的`Cell1D`对象不会从左边绕到右边，对于某些规则它在边界上创建了手工艺品 [?]。你可能想要使用`Wrap1D`，它是`Cell1D`的子类。 它也在`Cell1D.py`中定义。

练习 2

1990 年，Bak，Chen 和 Tang 提出了一种细胞自动机，它是一种森林火灾的抽象模型。 每个细胞处于三种状态之一：空，被树占用或着火。

CA 的规则是：

- 空细胞以概率`p`被占用。
- 如果任何一个邻居着火，那么带有树的细胞就会燃烧。
- 即使没有邻居着火，带有树的细胞自发燃烧，概率为`f`。
- 在下一个时间步骤中，着火的细胞变为空细胞。

编写一个实现这个模型的程序。 你可能想要继承`Cell2D`。 参数的常用值为`p = 0.01`和`f = 0.001`，但你可能想要尝试其他值。

从随机初始条件开始，运行 CA 直到它达到稳定状态，树的数量不再持续增加或减少。

在稳定状态下，森林分形的几何形状是什么？ 它的分形维度是多少？

八、自组织临界

> 原文：Chapter 8 Self-organized criticality
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

在前一章中，我们看到了一个具有临界点的系统的例子，并且我们探索了临界系统 - 分形几何的一个共同特性。

在本章中，我们将探讨临界系统的另外两个性质：重尾分布，我们在第五章中见过，和粉红噪声，我将在本章中解释。

这些性质是部分有趣的，因为它们在自然界中经常出现；也就是说，许多自然系统会产生分形几何，重尾分布和粉红噪声。

这个观察提出了一个自然的问题：为什么许多自然系统具有临界系统的特性？一个可能的答案是自组织临界性（SOC），这是一些系统向临界状态演化并保持它的趋势。

在本章中，我将介绍沙堆模型，这是第一个展示 SOC 的系统。

本章的代码位于本书仓库的`chap08.ipynb`中。使用代码的更多信息，请参见第？章。

8.1 临界系统

许多临界系统表现出常见的行为：

- 分形几何：例如，冷冻的水倾向于形成分形图案，包括雪花和其他晶体结构。分形的特点是自相似性；也就是说，图案的一部分与整体的缩放副本相似。
- 一些物理量的重尾分布：例如，在冷冻的水中，晶体尺寸的分布是幂律的。
- 呈现粉红噪声的时间变化。复合信号可以分解为它们的频率分量。在粉红噪声中，低频分量比高频分量功率更大。具体而言，频率`f`处的功率与`1 / f`成正比。

临界系统通常不稳定。例如，为了使水保持部分冷冻状态，需要主动控制温度。如果系统接近临界温度，则小型偏差倾向于将系统从一个相位移到另一个相位。

许多自然系统表现出典型的临界性行为，但如果临界点不稳定，它们本质上不应该是常见的。这是 Bak，Tang 和 Wiesenfeld 的解决的困惑。他们的解决方案称为自组织临界（SOC），其中“自组织”意味着从任何初始状态开始，系统都会转向临界状态，并停留在那里，无需外部控制。

8.2 沙堆

沙堆模型由 Bak，Tang 和 Wiesenfeld 于 1987 年提出。它不是一个现实的沙堆模型，而是一个抽象，它用（1）大量（2）与邻居互动的元素来模拟物理系统。

沙堆模型是一个二维细胞自动机，每个细胞的状态代表沙堆的部分斜率。在每个时间步骤中，检查每个细胞来查看它是否超过临界值 `K`，通常是 3。如果是，则它会“倒塌”并将沙子转移到四个相邻细胞；也就是说，细胞的斜率减少 4，并且每个邻居增加 1。在网格的周边，所有细胞保持为斜率 0，所以多余的会溢出边缘。

Bak，Tang 和 Wiesenfeld 首先将所有细胞初始化为大于 `K` 的水平，然后运行模型直至稳定。然后他们观察微小扰动的影响；他们随机选择一个细胞，将其值增加 1，然后再次运行模型，直至稳定。

对于每个扰动，他们测量 `T`，这是沙堆稳定所需的时间步数，`S` 是倒塌的细胞总数 [1]。

> [1] 原始论文使用了 `S` 的不同定义，但是后来的工作使用了这个定义。

大多数情况下，放置一粒沙子不会导致细胞倒塌，因此 `T = 1` 和 `S = 0`。 但偶尔一粒沙子会引起雪崩，影响很大一部分网格。 结果表明，`T` 和 `S` 的分布是重尾的，这支持了系统处于临界状态的断言。

他们得出结论：沙堆模型表现出“自组织临界性”，这意味着从最初的状态开始，它不需要外部控制，或者称之为“微调”任何参数，就可以向临界状态发展。 随着更多沙粒的添加，模型仍处于临界状态。

在接下来的几节中，我复制他们的实验并解释结果。

8.3 实现沙堆

为了实现沙堆模型，我定义了一个名为 `SandPile` 的类，该类继承自 `Cell2D.py` 中定义的 `Cell2D`。

```
class SandPile(Cell2D):

    def __init__(self, n, m, level=9):
        self.array = np.ones((n, m)) * level
```

数组中的所有值都初始化为 `level`，这通常大于倒塌阈值 `K`。

以下是 `step` 方法，它找到大于 `K` 的所有细胞并将它们推翻：

```
kernel = np.array([[0, 1, 0],
                   [1,-4, 1],
                   [0, 1, 0]], dtype=np.int32)

def step(self, K=3):
    toppling = self.array > K
    num_toppled = np.sum(toppling)
    c = correlate2d(toppling, self.kernel, mode='same')
    self.array += c
    return num_toppled
```

为了解释这是如何工作的，我将从一小堆开始，只有两个准备推翻的细胞：

```
pile = SandPile(n=3, m=5, level=0)
pile.array[1, 1] = 4
pile.array[1, 3] = 4
```

最初，`pile.array` 是这样：

```
[[0 0 0 0 0]
 [0 4 0 4 0]
 [0 0 0 0 0]]
```

现在我们可以选择高于倒塌阈值的细胞：

```
toppling = pile.array > K
```

结果是一个布尔数组，但是我们可以像整数数组一样使用它：

```
[[0 0 0 0 0]
 [0 1 0 1 0]
 [0 0 0 0 0]]
```

如果我们关联这个数组和核起来，它会在每个 `toppling` 是 1 的地方复制这个核。

```
c = correlate2d(toppling, kernel, mode='same')
```

这就是结果：

```
[[ 0  1  0  1  0]
 [ 1 -4  2 -4  1]
 [ 0  1  0  1  0]]
```

注意，在核的副本重叠的地方，它们会相加。

这个数组包含每个细胞的改变量，我们用它来更新原始数组：

```
pile.array += c
```

这就是结果：

```
[[0 1 0 1 0]
 [1 0 2 0 1]
 [0 1 0 1 0]]
```

这就是`step`的工作原理。

默认情况下，`correlate2d`认为数组的边界固定为零，所以任何超出边界的沙粒都会消失。

`SandPile`还提供了`run`，它会调用`step`，直到没有更多的细胞倒塌：

```
def run(self):
    total = 0
    for i in itertools.count(1):
        num_toppled = self.step()
        total += num_toppled
        if num_toppled == 0:
            return i, total
```

返回值是一个元组，其中包含时间步数和倒塌的细胞总数。

如果你不熟悉`itertools.count`，它是一个无限生成器，它从给定的初始值开始计数，所以`for`循环运行，直到`step`返回 0。

最后，`drop`方法随机选择一个细胞并添加一粒沙子：

```
def drop(self):
    a = self.array
    n, m = a.shape
    index = np.random.randint(n), np.random.randint(m)
    a[index] += 1
```

让我们看一个更大的例子，其中`n=20`：

```
pile = SandPile(n=20, level=10)
pile.run()
```

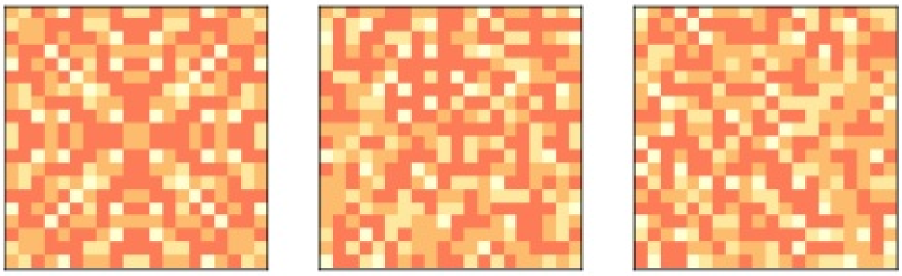


图 8.1：沙堆模型的初始状态（左），和经过 200 步（中）和 400 步（右）之后的状态

初始级别为 10 时，这个沙堆需要 332 个时间步才能达到平衡，共有 53,336 次倒塌。 图？（左）展示了初始运行后的状态。 注意它具有重复元素，这是分形特征。 我们会很快回来的。

图？（中）展示了在将 200 粒沙子随机放入细胞之后的沙堆构造，每次都运行直至达到平衡。 初始状态的对称性已被打破；状态看起来是随机的。

最后图？（右）展示了 400 次放置后的状态。 它看起来类似于 200 次之后的状态。 事实上，这个沙堆现在处于稳定状态，其统计属性不会随着时间而改变。 我将在下一节中解释一些统计属性。

8.4 重尾分布

如果沙堆模型处于临界状态，我们希望为一些数量找到重尾分布，例如雪崩的持续时间和大小。 所以让我们来看看。

我会生成一个更大的沙堆，`n = 50`，初始水平为 30，然后运行直到平衡状态：

```
pile2 = SandPile(n=50, level=30)
pile2.run()
```

下面我们会运行 100,000 个随机放置。

```
iters = 100000
res = [pile2.drop_and_run() for _ in range(iters)]
```

顾名思义，`drop_and_run` 调用 `drop` 和 `run`，并返回雪崩的持续时间和倒塌的细胞总数。

所以 `res` 是 `(T, S)` 元组的列表，其中 `T` 是持续时间，以时间步长为单位，并且 `S` 是倒塌的细胞。我们可以使用 `np.transpose` 将 `res` 解构为两个 NumPy 数组：

```
T, S = np.transpose(res)
```

大部分放置的持续时间为 1，没有倒塌的细胞，所以我们会将这些过滤掉。

```
T = T[T>1]
S = S[S>0]
```

`T` 和 `S` 的分布有许多小值和一些非常大的值。我将使用 `thinkstats2` 中的 `Hist` 类创建值的直方图；即每个值到其出现次数的映射。

```
from thinkstats2 import Hist

histT = Hist(T)
histS = Hist(S)
```

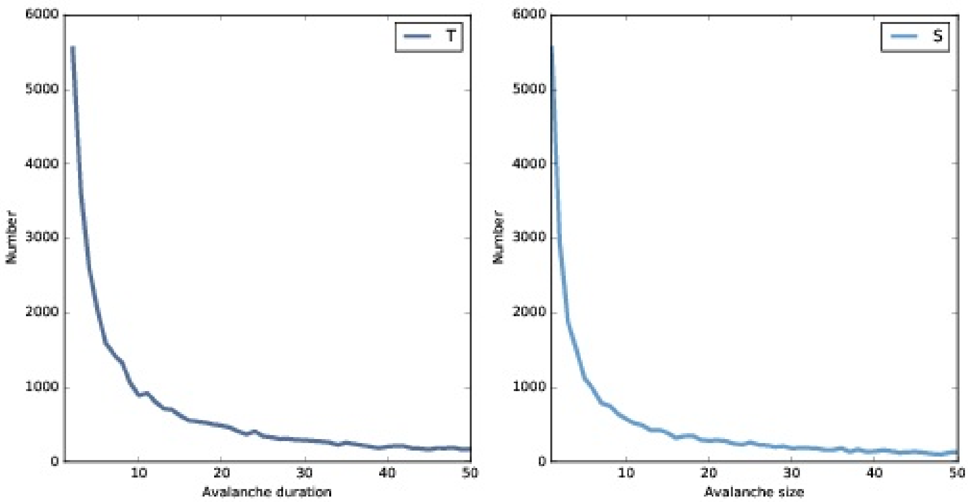


图 8.2：雪崩持续时间（左）和大小（右）的分布，线性刻度。

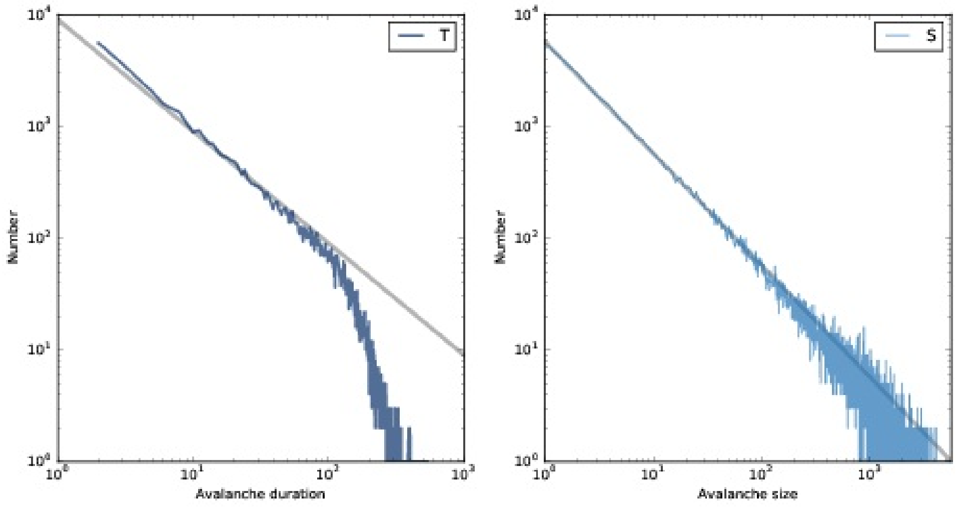


图 8.3：雪崩持续时间（左）和大小（右）的分布，双对数刻度。

图？显示了值小于 50 的结果。但正如我们在第？节中看到的那样，我们可以通过将它们绘制在双对数坐标上，更清楚地了解这些分布，如图？所示。

对于 1 到 100 之间的值，分布在双对数刻度上几乎是直的，这表示了重尾。 图中的灰线斜率为 -1，这表明这些分布遵循参数为 `α=1` 的幂律。

对于大于 100 的值，分布比幂律模型下降更快，这意味着较大值比模型的预测更少。 一种解释是，这种效应是由于沙堆的有限尺寸造成的，因此我们可能预计，更大的沙堆能更好地拟合幂律。

另一种可能性是，这些分布并不严格遵守幂律，你可以在本章结尾的一个练习中探索。 但即使它们不是幂律分布，它们仍然是重尾的，因为我们预计系统处于临界状态。

8.5 分形

临界系统的另一个属性是分形几何。 图中的初始状态 （左）类似于分形，但是你不能总是通过观察来分辨。 识别分形的更可靠的方法是估计其分形维度，正如第？节那样。

我首先制作一个更大的沙堆，`n = 131`，初始水平为 22。

```
pile3 = SandPile(n=131, level=22)
pile3.run()
```

顺便说一下，这个沙堆达到平衡需要 28,379 步，超过 2 亿个细胞倒塌。

为了更清楚地看到生成的团，我选择水平为 0, 1, 2 和 3 的细胞，并分别绘制它们：

```
def draw_four(viewer, vals=range(4)):
    thinkplot.preplot(rows=2, cols=2)
    a = viewer.viewee.array

    for i, val in enumerate(vals):
        thinkplot.subplot(i+1)
        viewer.draw_array(a==vals[i], vmax=1)
```

`draw_four` 需要 `SandPileViewer` 对象，它在 `Sand.py` 中定义。 参数 `vals` 是我们想要绘制的值的列表; 默认值是 0, 1, 2 和 3。

以下是它的使用方式：

```
viewer3 = SandPileViewer(pile3)
draw_four(viewer3)
```

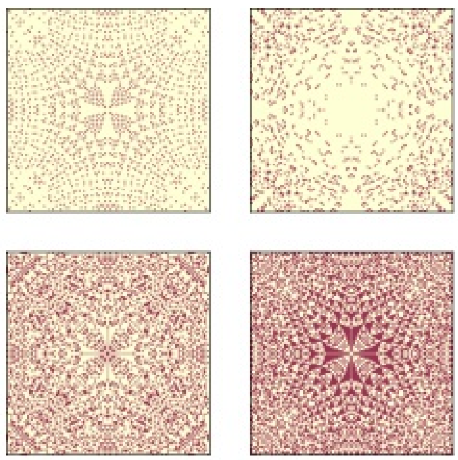


图 8.4：沙堆模型的初始状态，从上到下，从左到右选择水平为 0, 1, 2 和 3 的细胞。

图？展示了结果。 现在对于这些图案中的每一个，我们都可以使用方框计数算法来估计分形维数：我们将计算沙堆中心的小方框中的细胞数量，然后看看细胞数量随着方框变大而如何增加。 这是我的实现：

```
def count_cells(a):
    n, m = a.shape
    end = min(n, m)

    res = []
    for i in range(1, end, 2):
        top = (n-i) // 2
        left = (m-i) // 2
        box = a[top:top+i, left:left+i]
        total = np.sum(box)
        res.append((i, i**2, total))

    return np.transpose(res)
```

参数`a`是 NumPy 布尔数组，值为 0 和 1。 方框的大小最初为 1，每次循环中，它会增加 2，直到到达终点，它是`n`和`m`中较小的一个。

每次循环中，`box`都是一组宽度和高度为`i`的细胞，位于数组中心。`total`是方框中“开”细胞的数量。

结果是一个元组列表，其中每个元组包含`i`和`i ** 2`，用于比较，以及方框中的细胞数量。

最后，我们使用`np.transpose`生成一个 NumPy 数组，其中包含`i`，`i ** 2`和`total`。

为了估计分形维度，我们提取行：

```
steps, steps2, cells = res
```

之后绘制结果：

```
thinkplot.plot(steps, steps2, linestyle='dashed')
thinkplot.plot(steps, cells)
thinkplot.plot(steps, steps, linestyle='dashed')
```

然后使用`linregress`在双对数刻度上拟合直线：

```
from scipy.stats import linregress

params = linregress(np.log(steps), np.log(cells))
slope = params[0]
```

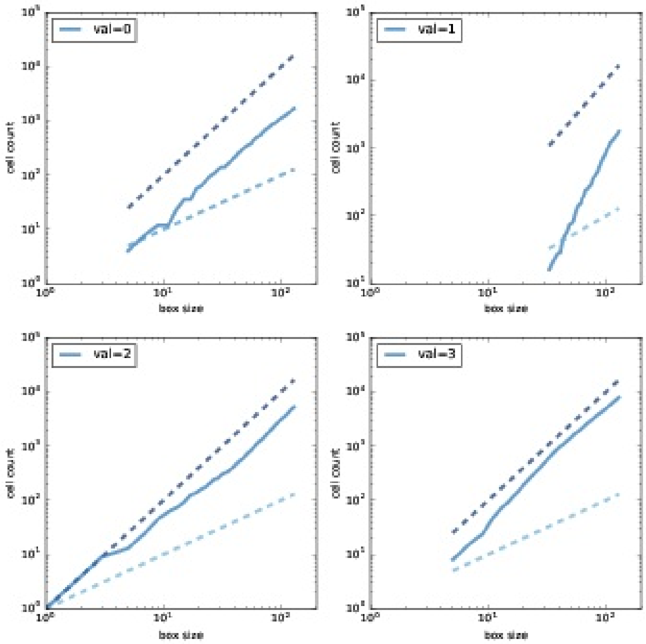


图 8.5：与斜率位 1 和 2 的虚线相比，水平为 0, 1, 2 和 3 的细胞的方框计数。

图？展示了结果。 请注意，只有 `val = 2` （左下）从方框大小 1 开始，因为中心细胞的值为 2；其他直线从包含非零细胞数的第一个方框大小开始。

在双对数刻度上，细胞计数几乎形成直线，这表明我们正在测量方框大小的有效范围内的分形维度。

估计的分形维度是：

```
0  1.871
1  3.502
2  1.781
3  2.084
```

值为 0, 1 和 2 的分形维度似乎明显不是整数，这表明图像是分形的。

严格来说，值为 3 的分形维度与 2 不可区分，但考虑到其他值的结果，线的表观曲率和图案的外观，似乎它也是分形的。

本章结尾的练习之一，要求你使用不同的 `n` 和 `level` 值，再次运行此分析，来查看估计的维度是否一致。

8.6 频谱密度

提出沙堆模型的原始论文的标题是《自组织临界：1/f 噪声的解释》（Self-Organized Criticality: An Explanation of 1/f Noise）。 正如小标题所述的那样，Bak，Tang 和 Wiesenfeld 正试图解释为什么许多自然和工程系统表现出 1/f 噪声，这也被称为“闪烁噪声”和“粉红噪声”。

为了了解粉红噪声，我们必须绕路来了解信号，频谱分析和噪声。

信号：

信号是随时间变化的任何数量。 一个例子是声音，即空气密度的变化。 在本节中，我们将探讨雪崩持续时间和大小在不同时间段内的变化。

频谱分析：

任何信号都可以分解为一组具有不同音量或功率的频率分量。 例如，演奏中央 C 上方的 A 的小提琴的声音，包含频率为 440 Hz 的主要分量，但它也包含较低功率分量，例如 880 Hz，1320 Hz 和其他整数倍的基频。 频谱分析是寻找构成信号的成分和它们的功率的过程，称为其频谱。

噪声：

在通常的用法中，“噪声”通常是一种不需要的声音，但在信号处理的情况下，它是一个包含许多频率成分的信号。

噪声有很多种。 例如，“白噪声”是一个信号，它在很宽的频率范围内拥有相同功率的成分。

其他种类的噪声在频率和功率之间有不同的关系。 在“红噪声”中，频率为 `f` 的功率为 `1 / f ** 2`，我们可以这样写：

```
P(f) = 1 / f ** 2
```

我们可以把指数 2 换成 `β` 来扩展它：

```
P(f) = 1 / f ** β
```

当 `β= 0` 时，该等式描述白噪声; 当 `β= 2` 时，它描述红噪声。 当参数接近 1 时，我们将结果称为 `1 / f` 噪声。 更一般来说，任何介于 0 和 2 之间的噪声称为“粉红”，因为它介于白色和红色之间。

那么这如何适用于沙堆模型呢？ 假设每次细胞倒塌时，它会发出声音。 如果我们在运行中记录了沙堆模型，它会是什么样子？

在我的 `SandPile` 实现运行时，它会记录在每个时间步骤中，倒塌的细胞数量，并将结果记录在名为 `toppled_seq` 的列表中。 例如，在第？节中运行模型之后，我们可以提取产生的信号：

```
signal = pile2.toppled_seq
```

为了计算信号的频谱（同样，这是它包含的频率和它们的功率），我们可以使用快速傅立叶变换（FFT）。

唯一的问题是噪声信号的频谱往往是嘈杂的。 但是，我们可以通过将一个长信号分成多个段，计算每个段的 FFT，然后计算每个频率的平均功率来使其平滑。

该算法的一个版本被称为“韦尔奇方法”，SciPy 提供了一个实现。 我们可以像这样使用它：

```python
from scipy.signal import welch

nperseg = 2048
freqs, spectrum = welch(signal, nperseg=nperseg, fs=nperseg)
```

`nperseg` 是信号分解成的片段长度。 对于较长的片段，我们可以获得更多的频率成分，但由于平均的片段数较少，结果更加嘈杂。

`fs` 是“采样频率”，即每单位时间的信号中的数据点数。 通过设置 `fs = nperseg`，我们可以得到从 0 到 `nperseg / 2` 的频率范围，但模型中的时间单位是任意的，所以频率并不意味着什么。

返回值，`freqs` 和 `powers` 是 NumPy 数组，包含成分的频率及其相应的功率。

如果信号是粉红噪声，我们预计：

```
P(f) = 1 / f ** β
```

对两边取对数会得到：

```
logP(f) = −β logf
```

所以如果我们在双对数刻度上绘制功率与频率的关系，我们预计有一条斜率为 `β` 的直线。

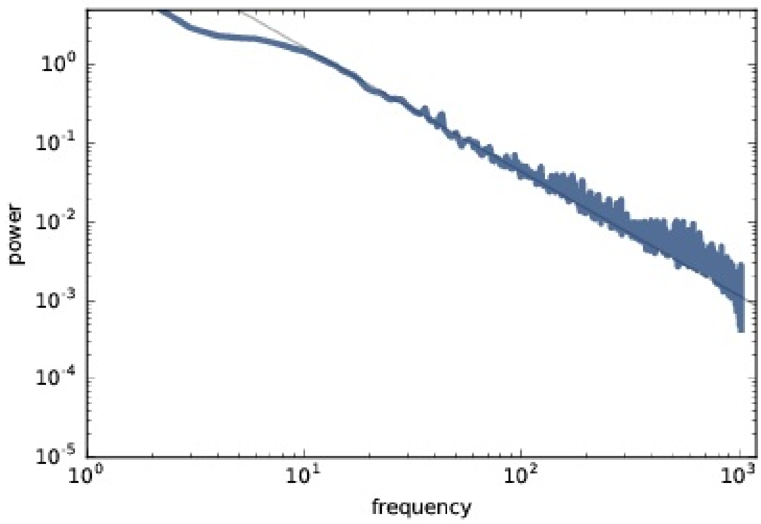


图 8.6：随着时间推移的倒塌细胞的功率频谱，双对数刻度

图？显示结果。 对于 10 到 1000 之间的频率（以任意单位），频谱落在一条直线上。 灰线斜率为 -1.58，这对应于由 Bak，Tang 和 Wiesenfeld 报告的参数 `β= 1.58` 。

这个结果证实了沙堆模型产生粉红噪声。

8.7 还原论与整体论

Bak，Tang 和 Wiesenfeld 的原始论文是过去几十年中被引用次数最多的论文之一。 其他系统已被证明是自组织临界，特别是沙堆模型已被详细研究。

事实证明，沙堆模型并不是一个很好的沙堆模型。 沙子密集且不太粘，所以动量对雪崩的行为有着不可忽视的影响。 因此，与模型预测的相比，非常大和非常小的雪崩数量更少，并且分布可能不是重尾。

Bak 建议，这个观察没有考虑到这一点。 沙堆模型并是现实的沙堆模型；它旨在成为一大类系统的简单模型。

为了理解这个要点，考虑两种模式，还原论和整体论，会有帮助。 还原论模型通过描述系统的各个部分及其相互作用来描述系统。 当还原论模型用于解释时，它取决于模型和系统组成部分之间的类比。

例如，为了解释为什么理想气体定律成立，我们可以用点质量模拟组成气体的分子，并将它们的相互作用建模为弹性碰撞。 如果你模拟或分析这个模型，你会发现它服从理想的气体定律。 一定程度上，该模型满足了，气体中的分子表现为模型中分子。 类比位于系统的各个部分和模型的各个部分之间。

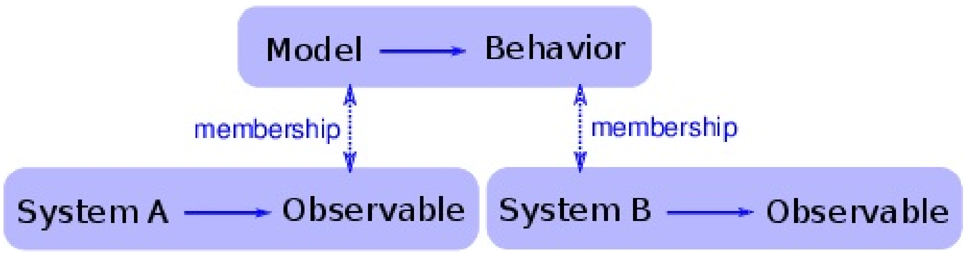


图 8.7：还原论模型的逻辑结构

整体论模型更关注系统之间的相似性，而对类比部分不太感兴趣。 整体论建模方法通常由两个步骤组成，不一定按以下顺序：

- 识别出现在各种系统中的一种行为。
- 寻找证明这种行为的简单模型。

例如，在《自私的基因》（The Selfish Gene）中，理查德道金斯（Richard Dawkins）认为，遗传进化只是进化系统的一个例子。 他确定了这些类别的基本元素 - 离散复制器（discrete replicator），可变性和差异生殖（differential reproduction） - 并提出任何包含这些元素的系统都会显示类似的行为，包括不带设计的复杂性。

作为演化系统的另一个例子，他提出了“模因”（memes），即通过人与人之间的传播而复制的思想或行为 [2]。 由于模因争夺人类的注意力的资源，它们的演变方式与遗传进化相似。

> [2] “模因”的用法源自 Dawkins，并且早于 20 年前这个词在互联网上的衍生用法。

模因模型的批评者指出，模因是基因的一个很差的类比。 模因在许多方面与基因不同。 但道金斯认为，这些差异并不重要，因为模因不可能与基因类似。 相反，模因和基因是同一类别的例子：进化系统。 它们之间的差异强调了真正的观点，即进化是一个适用于许多看似不同的系统的通用模型。 图？显示了这个论述的逻辑结构。

Bak 也提出了类似的观点，即自组织临界性是一大类系统的通用模型：

> 由于这些现象无处不在，它们不能依赖于任何具体的细节......如果一大类问题的物理结果是相同的，这为【理论家】提供了选项，选择属于该类的最简单的可能的【模型】，来进行详细的研究 [3]。
>
> [3] Bak, How Nature Works, Springer-Verlag 1996, page 43.

许多自然系统表现出临界系统的行为特征。 Bak 对这种普遍性的解释是，这些系统是自组织临界性的示例。 有两种方式可以支持这个论点。 一个是建立一个特定系统的现实模型，并显示该模型表现出 SOC。 其次是证明 SOC 是许多不同模型的一个特征，并确定这些模型共同的基本特征。

第一种方法我称之为还原论，可以解释特定系统的行为。 第二种是整体论的方法，解释了自然系统中临界性的普遍性。 他们是不同目的的不同模型。

对于还原论模型，现实主义是主要优点，简单性是次要的。 对于整体模型，这是相反的。

8.8 SOC，因果和预测

如果股市指数在一天内下跌一个百分点，则不需要解释。 但如果它下降 10%，人们想知道为什么。 电视上的专家愿意提供解释，但真正的答案可能是没有解释。

股票市场的日常变化展示了临界性的证据：数值变化的分布是重尾的，时间序列表现出粉红噪音。 如果股票市场是一个临界系统，我们应该预计，偶尔的巨大变化是市场普通行为的一部分。

地震强度的分布也是重尾的，并且存在可能解释这种行为的，地质断层动力学的简单模型。如果这些模型是正确的，那就意味着大地震是普遍的； 也就是说，与小地震相比，他们不需要更多解释。

同样，查尔斯·佩罗（Charles Perrow）认为，像核电厂这样的大型工程系统的故障，就像沙堆模型中的雪崩一样。 大多数故障是小的，孤立的和无害的，但偶尔坏运气的巧合会产生灾难。 当发生重大事故时，调查人员会去寻找原因，但如果佩罗的“正常事故理论”是正确的，那么可能没有特殊原因导致严重故障。

这些结论并不令人欣慰。 除其他事情外，它们意味着大地震和某些事故从根本上是不可预测的。不可能观察临界系统的状态，并说大雪崩是否“来临”。 如果系统处于临界状态，则总是可能发生大型雪崩。 它只取决于下一粒沙子。

在沙堆模型中，大雪崩的原因是什么？哲学家有时会将直接（proximate）原因与根本（ultimate）原因区分开来，前者是最直接的原因，无论出于何种原因，后者被视为真正的原因。

在沙堆模型中，雪崩的直接原因是一粒沙子，但引起大雪崩的沙粒与其他沙粒相同，所以它没有提供特别的解释。大雪崩的根本原因是整个系统的结构和动态：大雪崩的发生是因为它们是系统的一个属性。

许多社会现象，包括战争，革命，流行病，发明和恐怖袭击，其特点是重尾分布。如果这些分布的原因是社会系统是临界的，那么这表明主要的历史事件可能从根本上是不可预测的，并且无法解释。

8.9 练习

练习 1

本章的代码位于本书仓库中的 Jupyter 笔记本 `chap08.ipynb` 中。打开这个笔记本，阅读代码，然后运行单元格。你可以使用这个笔记本来练习本章的练习。我的解决方案在 `chap08soln.ipynb` 中。

练习 2

为了检验 `T` 和 `S` 的分布是否是重尾的，我们将它们的直方图绘制在双对数刻度上，这就是 Bak，Tang 和 Wiesenfeld 在他们的论文中所展示。但我们在第？节中看到，这种可视化可能掩盖分布的形状。使用相同的数据，绘制一个图表，显示 `S` 和 `T` 的累积分布（CDF）。对于他们的形状你可以说什么？他们是否遵循幂律？他们是重尾的嘛？

你可能会发现将 CDF 绘制在对数和双对数刻度上会有所帮助。

练习 3

在第？章，我们发现沙堆模型的初始状态会产生分形图案。但是在我们随机放置了大量的沙粒之后，图案看起来更随机。

从第？章中的示例开始，运行沙堆模型一段时间，然后计算 4 个水平中的每个的分形维度。沙堆模型是否处于稳定状态？

练习 4

另一种沙堆模型，称为“单一来源”模型，从不同的初始条件开始：中心细胞设为较大值，除了中心细胞，所有细胞设为 0，而不是所有细胞都是同一水平的。编写一个创建 `SandPile` 对象的函数，设置单一来源的初始条件，并运行，直到达到平衡。结果出现了分形吗？

你可以在 http://math.cmu.edu/~wes/sandgallery.html 上了解这个版本的沙堆模型。

练习 5

在 1989 年的一篇论文中，Bak，Chen 和 Creutz 认为生命游戏是一个 SOC 系统 [5]。

> [5] “Self-organized criticality in the Game of Life”，可以在这里获取：http://www.nature.com/nature/journal/v342/n6251/abs/342780a0.html。

为了复制他们的测试，运行 GoL CA 直到稳定，然后随机选择一个细胞并翻转它。运行 CA 直到它再次稳定下来，跟踪 `T`，这个是它需要的时间步数，以及 `S`，受影响的细胞数。重复进行大量试验并绘制 `T` 和 `S` 的分布。同时，记录在每个时间步骤中改变状态的细胞数量，并查看得到的时间序列是否与粉红噪声相似。

练习 6

在《自然界的分形几何》（The Fractal Geometry of Nature）中，Benoit Mandelbrot 提出了自然系统中重尾分布的普遍性的解释，他称之为“异端”。 正如 Bak 所言，许多系统可以独立产生这种行为。 相反，它们可能只是少数，但系统之间可能会有交互，它们导致行为的传播。

为了支持这个论点，Mandelbrot 指出：

- 观测数据的分布通常是“一个固定的底层真实分布，和一个高度可变的过滤器的联合效应”。
- 重尾分布对于过滤器是健壮的；也就是说，“各种过滤器使其渐近行为保持不变”。

你怎么看待这个观点？ 你会把它描述为还原论还是整体论？

练习 7

在 http://en.wikipedia.org/wiki/Great_man_theory 上阅读“伟人”的历史理论。 自组织临界对这个理论有什么暗示？

九、基于智能体的模型

> 原文：Chapter 9 Agent-based models
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

我们迄今为止看到的模型可能具有“基于规则”的特征，因为它们涉及受简单规则支配的系统。在本章和以后的章节中，我们将探索基于智能体（agent）的模型。

基于智能体的模型包含智能体，它旨在模拟人和其他实体，它们收集世界的信息，制定决策并采取行动。

智能体通常位于空间或网络中，并在本地彼此交互。 他们通常有不完整的，不全面的世界信息。

智能体之间经常存在差异，而不像以前的所有模型，它们的所有成分都相同。 基于智能体的模型通常包含智能体之间，或世界中的随机性。

自 20 世纪 70 年代以来，基于智能体的模型已成为经济学和其他社会科学，以及一些自然科学中的重要工具。

基于智能体的模型对非均衡系统的动态建模（尽管它们也用于研究均衡系统）非常有用。 它们对于理解个人决策和系统行为之间的关系特别有用。

本章的代码位于`chap09.ipynb`中，它是本书仓库中的 Jupyter 笔记本。 使用此代码的更多信息，请参见第？节。

9.1 谢林模型

1971 年，托马斯谢（Thomas Schelling）发表了《隔离的动态模型》（Dynamic Models of Segregation），该模型提出了种族隔离的简单模型。 谢林模型的世界是一个网格；每个细胞代表一栋房子。 房屋被两种智能体占用，标记为红色和蓝色，数量大致相同。 大约 10% 的房屋是空的。

在任何时候，智能体可能会高兴或不高兴，这取决于领域中的其他智能体，每个房屋的“邻居”是八个相邻细胞的集合。在一个版本的模型中，如果智能体至少有两个像他们一样的邻居，智能体会高兴，但如果是一个或零，他们就会不高兴。

模拟的过程是，随机选择一个智能体并检查他们是否高兴。 如果是这样，没有任何事情发生。如果不是，智能体随机选择其中一个未占用的细胞并移动。

听到这种模型导致一些隔离，你可能不会感到惊讶，但是你可能会对这个程度感到惊讶。 很快，会出现相似智能体的群落。 随着时间的推移，这些群落会不断聚合，直到有少量的大型群落，并且大多数智能体生活在同质社区中。

如果你不知道这个过程，只看到结果，你可能会认为智能体是种族主义者，但实际上他们都会在一个混合的社区感到非常高兴。 由于他们不愿意数量过大，所以在最坏的情况下，他们可能被认为是排外的。 当然，这些智能体是真实人物的过度简化，所以这些描述可能根本不恰当。

种族主义是一个复杂的人类问题; 很难想象这样简单的模型可以揭示它。 但实际上，它提供了一个强有力论据，有关系统及其各部分之间关系的：如果你观察真实城市的隔离，你不能总结为，个人的种族主义是直接原因，或者，城市居民是种族主义者。

当然，我们必须牢记这个论述的局限性：谢林模型证明了隔离的一个可能原因，但没有提到实际原因。

9.2 谢林模型的实现

为了实现谢林模型，我编写了另一个继承`Cell2D`的类：

```
class Schelling(Cell2D):

    def __init__(self, n, m=None, p=0.5):
        self.p = p
        m = n if m is None else m
        choices = [0, 1, 2]
        probs = [0.1, 0.45, 0.45]
        self.array = np.random.choice(choices, (n, m), p=probs)
```

参数`n`和`m`是网格的维度，`p`是相似邻居比例的阈值。 例如，如果`p = 0.5`，也就是其邻居中少于 50% 为相同颜色，则智能体将不高兴。

`array`是 NumPy 数组，其中每个细胞如果为空，则为 0；如果由红色智能体占用，则为1；如果由蓝色智能体占用，则为 2。 最初，10% 的细胞是空的，45% 为红色和 45% 为蓝色。

谢林模型的`step`函数比以前的`step`函数复杂得多。 如果你对细节不感兴趣，你可以跳到下一节。 但是如果你坚持要看，你可能需要一些 NumPy 的提示。

首先，我将生成逻辑数组，表明哪些细胞是红色，蓝色和占用的：

```
a = self.array
red = a==1
blue = a==2
occupied = a!=0
```

我将使用 `np.correlate2d` 来计算，对于每个细胞，红色相邻细胞的数量和被占用的细胞数量。

```
options = dict(mode='same', boundary='wrap')

kernel = np.array([[1, 1, 1],
                   [1, 0, 1],
                   [1, 1, 1]], dtype=np.int8)

num_red = correlate2d(red, kernel, **options)
num_neighbors = correlate2d(occupied, kernel, **options)
```

现在对于每个细胞，我们可以计算出红色的邻居比例和相同颜色的比例：

```
frac_red = num_red / num_neighbors
frac_blue = 1 - frac_red
frac_same = np.where(red, frac_red, frac_blue)
```

`frac_red` 只是 `num_red` 和 `num_neighbors` 的比率，而 `frac_blue` 是 `frac_red` 的补。

`frac_same` 有点复杂。 函数 `np.where` 就像逐元素的 `if` 表达式一样。 第一个参数是从第二个或第三个参数中选择元素的条件。

在这种情况下，如果 `red` 为 `True` ， `frac_same` 获取 `frac_red` 的相应元素。 在红色为 `False` 的情况下， `frac_same` 获取 `frac_blue` 的相应元素。

现在我们可以确定不满意的智能体的位置：

```
unhappy_locs = locs_where(occupied & (frac_same < self.p))
```

结果 `unhappy_locs` 是一个 NumPy 数组，其中每行都是占用的细胞的坐标，其中 `frac_same` 低于阈值 `p` 。

`locs_where` 是 `np.nonzero` 的包装函数：

```
def locs_where(condition):
    return np.transpose(np.nonzero(condition))
```

`np.nonzero` 接受一个数组并返回所有非零元素的坐标，但结果是两个元组的形式。 `np.transpose` 将结果转换为更有用的形式，即每行都是坐标对的数组。

同样， `empty_locs` 是一个数组，包含空细胞的坐标：

```
empty_locs = locs_where(a==0)
```

现在我们到达了模拟的核心。 我们遍历不高兴的智能体并移动它们：

```
for source in unhappy_locs:
    i = np.random.randint(len(empty_locs))
    dest = tuple(empty_locs[i])
    a[dest] = a[tuple(source)]
    a[tuple(source)] = 0
    empty_locs[i] = source
```

`i` 是一个用来随机选择空细胞的索引。

`dest` 是一个包含空细胞的坐标的元组。

为了移动智能体，我们将值从 `source` 复制到 `dest`，然后将 `source` 的值设置为 0（因为它现在是空的）。

最后，我们用 `source` 替换 `empty_locs` 中的条目，以便刚刚变为空的细胞可以由下一个智能体选择。

9.3 隔离

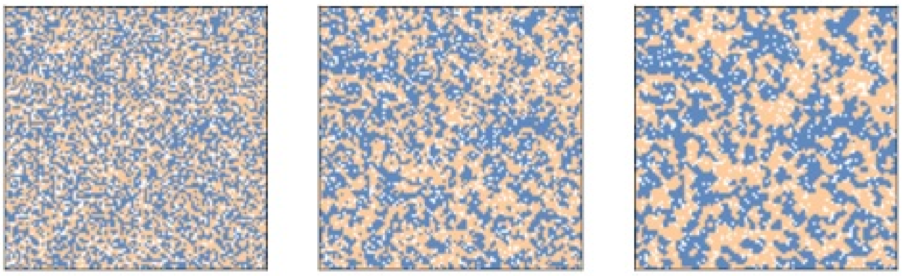


图 9.1：谢林的隔离模型，`n = 100`，初始条件（左），2 步后（中）和 10 步后（右）

现在让我们看看我们运行模型时会发生什么。 我将以 `n = 100` 和 `p = 0.3` 开始，并运行 10 个步骤。

```
grid = Schelling(n=100, p=0.3)
for i in range(10):
    grid.step()
```

图？展示了初始状态（左），2 步（中）后和 10 步（右）后的模拟。

群落迅速形成，红色和蓝色的智能体移动到隔离集群中，它们由空细胞的边界分隔。

对于每种状态，我们可以计算隔离度，它是相同颜色的邻居的比例。在所有细胞中的平均值：

```
np.sum(frac_same) / np.sum(occupied)
```

在图？中，相似邻居的比例均值在初始状态中为 55%，两步后为 71%，10 步后为 80%！

请记住，当`p = 0.3`时，如果 8 个邻居中的 3 个是他们自己的颜色，那么智能体会很高兴，但他们最终居住在一个社区中，其中 6 或 7 个邻居是自己的颜色。

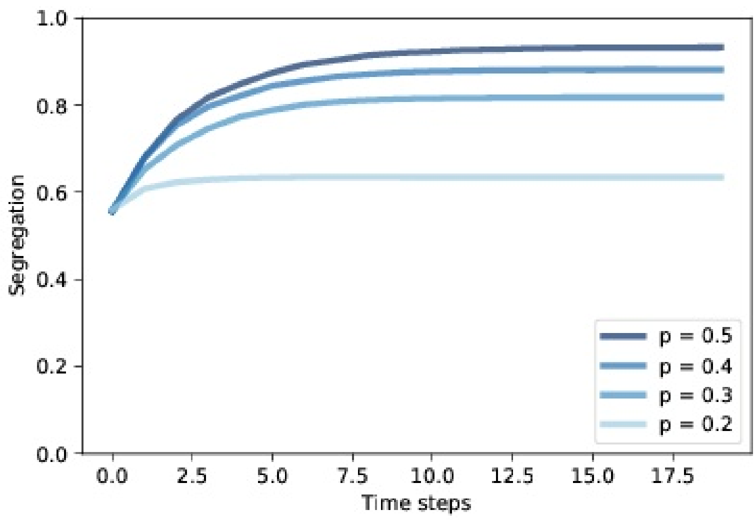


图 9.2：随着时间的推移，谢林模型中的隔离程度，范围为`p`

图？显示了隔离程度如何增加，以及它在几个`p`值下的平稳位置。 当`p = 0.4`时，稳定状态下的隔离程度约为 88%，且大多数智能体没有不同颜色的邻居。

这些结果令许多人感到惊讶，它们成为了个人决策与系统行为之间的，复杂且不可预测的关系的鲜明示例。

9.4 糖域

1996年，约书亚爱波斯坦（Joshua Epstein）和罗伯特阿克斯特尔（Robert Axtell）提出了糖域（Sugarscape），这是一个“人造社会”的智能体模型，旨在支持经济学和其他社会科学的相关实验。

糖域是一款多功能的模型，适用于各种主题。 作为例子，我将复制 Epstein 和 Axtell 的书《Growing Artificial Societies》的前几个实验。

糖域最简单的形式是一个简单的经济模型，智能体在二维网格上移动，收集和累积代表经济财富的“糖”。 网格的一些部分比其他部分产生更多的糖，并且一些智能体比其他人更容易找到它。

这个糖域的版本常用于探索和解释财富的分布，特别是不平等的趋势。

在糖域的网格中，每个细胞都有一个容量，这是它可容纳的最大糖量。 在原始状态中，有两个高糖区域，容量为 4，周围是同心环，容量分别为 3, 2 和 1。

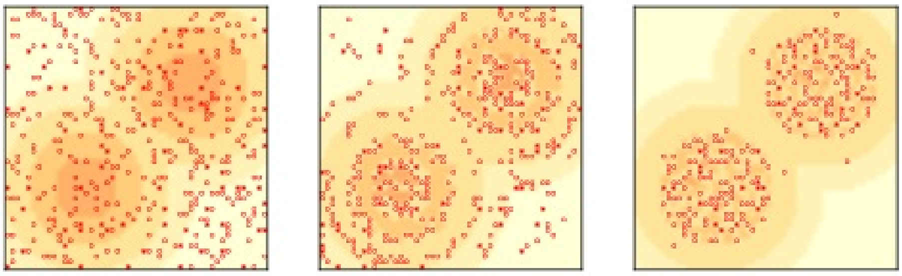


图 9.3：原始糖域模型的复制品：初始状态（左），2 步后（中）和 100 步后（右）。

图？（左）展示了初始状态，最黑暗的区域表示容量最高的细胞，小圆圈表示智能体。

最初有随机放置的 400 个智能体。 每个智能体有三个随机选择的属性：

糖：

每个智能体最开始都有先天的糖分，从 5 到 25 之间均匀选择。

代谢：

在每个时间步骤中，每个智能体都必须消耗一定数量的糖，从 1 到 4 之间均匀选择。

视力：

每个智能体可以“看到”附近细胞中糖量，并移动到最多的细胞，但是与其它智能体相比，一些智能体可以看到更远的细胞。 智能体看到的距离从 1 和 6 之间均匀选择。

在每个时间步骤中，智能体以随机顺序一次移动一格。 每个智能体都遵循以下规则：

- 智能体在 4 个罗盘方向的每一个方向上调查 `k` 个细胞，其中 `k` 是智能体的视野范围。
- 它选择糖分最多的未占用的细胞。 在相等的情况下，选择较近的细胞；在距离相同的细胞中，它随机选择。
- 智能体移动到选定的细胞并收获糖分，将收获增加到其积累的财富并将细胞清空。
- 智能体根据代谢消耗其财富的一部分。 如果结果总量为负数，智能体“饿死”并被删除。

在所有智能体完成这些步骤之后，细胞恢复一些糖，通常为 1 单位，但每个细胞中的总糖分受其容量限制。

图？（中）显示两步后模型的状态。 大多数智能体正在移到糖最多的地区。 视力高的智能体移动速度最快；视力低的智能体往往会卡在高原上，随机游走，直到它们足够接近来看到下一个水平。

出生在糖分最少的地区的智能体可能会饿死，除非他们的视力很好，先天条件也很高。

在高糖地区，随着糖分的出现，智能体相互竞争，寻找和收获糖分。 消耗高或视力低的智能体最有可能挨饿。

当糖在每个时间步骤增加 1 个单位时，就没有足够的糖来维持我们开始的 400 个智能体。 人口起初迅速下降，然后缓慢下降，在大约 250 左右停下。

图？（右）显示了 100 个时间步后的模型状态，大约有 250 个智能体。 存活的智能体往往是幸运者，出生时视力高和/或代消耗低。 存活到这里的话，它们可能会永远存活，积累无限量的糖。

9.5 财富的不平等

在目前的形式下，糖域建立了一个简单的生态学模型，可用于探索模型参数之间的关系，如增长率和智能体的属性，以及系统的承载能力（在稳定状态下生存的智能体数量）。 它模拟了一种形式的自然选择的，“适应度”较高的智能体更有可能生存下来。

该模型还表现出一种财富不平等，一些智能体积累糖的速度比其他智能体快。 但是对于财富分布，很难说具体的事情，因为它不是“静止的”。 也就是说，分布随着时间的推移而变化并且不会达到稳定状态。

然而，如果我们给智能体有限的寿命，这个模型会产生固定的财富分布。 然后我们可以运行实验，来查看参数和规则对此分布的影响。

在这个版本的模型中，智能体的年龄在每个时间步增加，并且从 60 到 100 之间的均匀分布中，随机选择一个寿命。如果智能体的年龄超过其寿命，它就会死亡。

当智能体因饥饿或年老而死亡时，它由属性随机的新智能体所取代，所以总人口是不变的。

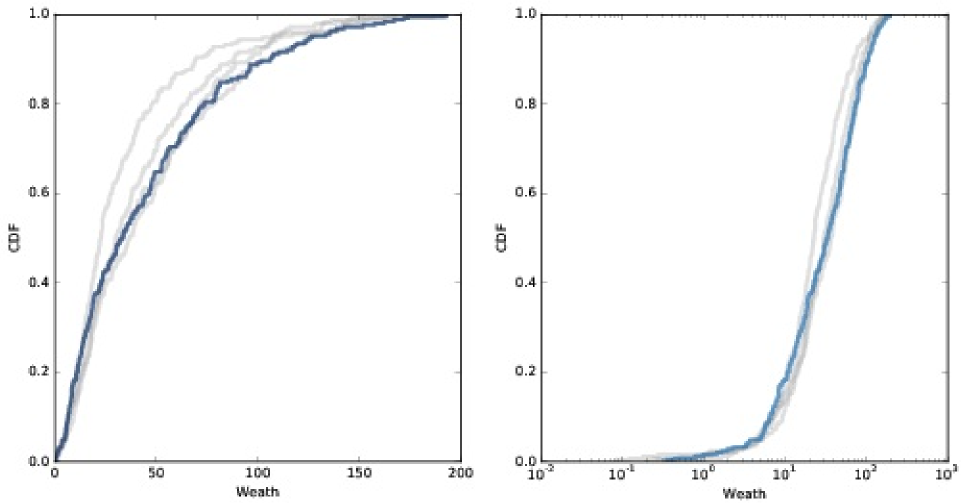


图 9.4：100, 200, 300 和 400 步（灰线）和 500 步（黑线）之后的糖（财富）的分布。 线性刻度（左）和对数刻度（右）。

从接近承载能力的 250 个智能体开始，我运行了 500 个步骤的模型。 在每 100 步之后，我绘制了智能体积累的糖的分布。 图？在线性刻度（左）和对数刻度（右）中展示结果。

经过大约 200 步（这是最长寿命的两倍）后，分布变化不大。 并且它向右倾斜。

大多数智能体积累的财富很少：第 25 百分位数大约是 10，中位数大约是 20。但是少数智能体积累了更多：第 75 百分位数是大约 40，最大值大于 150。

在对数刻度上，分布的形状类似于高斯或正态分布，但右尾被截断。 如果它在对数刻度上实际上是正态的，则分布是对数正态分布，这是一种重尾分布。 实际上，几乎每个国家和全世界的财富分布都是重尾分布。

如果说糖域解释了为什么财富分布是重尾的，但是糖域变化中的不平等的普遍性表明，不平等是许多经济体的特征，甚至是非常简单的经济体。 一些实验表明避免或减轻并不容易，它们带有一些规则，对纳税和其他收入转移进行建模。

9.6 实现糖域

糖域比以前的模型更复杂，所以我不会在这里介绍整个实现。 我将概述代码的结构，你可以在 Jupyter 笔记本`chap09.ipynb`中查看本章的细节，它位于本书的仓库中。 如果你对细节不感兴趣，你可以跳到下一节。

以下是带有`step`方法的`Agent`类：

```python
class Agent:

    def step(self, env):
        self.loc = env.look_around(self.loc, self.vision)
        self.sugar += env.harvest(self.loc) - self.metabolism
        self.age += 1
```

在每个步骤中，智能体移动，收获糖，并增加年龄。

参数 `env` 是环境的引用，它是一个 `Sugarscape` 对象。 它提供了方法 `look_around` 和收获：

- `look_around` 获取智能体的位置，这是一个坐标元组，以及智能体的视野，它是一个整数。 它返回智能体的新位置，这是糖分最多的可见细胞。
- `harvest` 需要智能体的（新）位置，并在移除并返回该位置的糖分。

这里是 `Sugarscape` 类和它的 `step` 方法（不需要替换）：

```python
class Sugarscape(Cell2D):

    def step(self):

        # loop through the agents in random order
        random_order = np.random.permutation(self.agents)
        for agent in random_order:

            # mark the current cell unoccupied
            self.occupied.remove(agent.loc)

            # execute one step
            agent.step(self)

            # if the agent is dead, remove from the list
            if agent.is_starving():
                self.agents.remove(agent)
            else:
                # otherwise mark its cell occupied
                self.occupied.add(agent.loc)

        # grow back some sugar
        self.grow()
        return len(self.agents)
```

`Sugarscape` 继承自 `Cell2D`，因此它与我们所见过的其他基于网格的模型相似。

这些属性包括 `agents`，它是 `Agent` 对象的列表，以及 `occupied`，它是一组元组，其中每个元组包含智能体占用的细胞的坐标。

在每个步骤中， `Sugarscape` 以随机顺序遍历智能体。 它调用每个智能体的 `step`，然后检查它是否已经死亡。 所有智能体都移动后，一些糖会恢复。

如果你有兴趣深入了解 NumPy，你可能需要仔细看看 `make_visible_locs`，它构建一个数组，其中每行包含智能体可见的细胞坐标，按距离排序，但距离相同的细胞 是随机顺序。

你可能想看看 `Sugarscape.make_capacity`，它初始化细胞的容量。 它演示了 `np.meshgrid` 的使用，这通常很有用，但需要一些时间才能理解。

9.7 迁移和波动行为

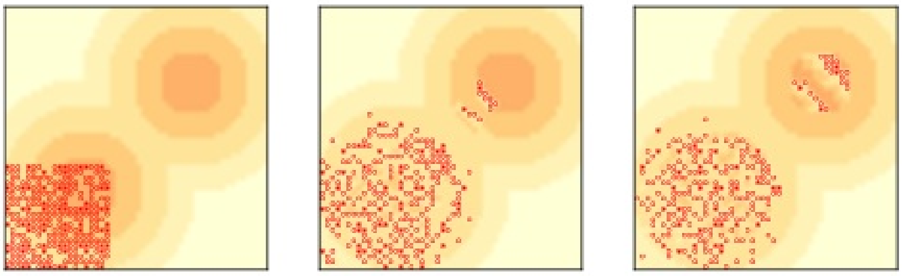


图 9.5：`Sugarscape`中的波动行为：初始状态（左），6 步后（中）和 12 步后（右）

虽然`Sugarscape`的主要目的不是探索空间中的智能体的移动，但 Epstein 和 Axtell 在智能体迁移时，观察到一些有趣的模式。

如果我们开始把所有智能体放在左下角，他们会迅速走向最近的高容量细胞的“山峰”。但是如果有更多的智能体，单个山峰不足以支持它们的话，他们很快就会耗尽糖分，智能体被迫进入低容量地区。

视野最长的那些，首先穿过山峰之间的山谷，并且像波一样向东北方向传播。因为他们在身后留下一些空细胞，所以其他智能体不会追随，直到糖分恢复。

结果是一系列离散的迁移波，每个波都像一个连贯的物体，就像我们在规则 110 CA 和生命游戏中看到的飞船（参见第？节）。

图？显示了初始条件下（左），6 个步骤（中）和 12 个步骤（右）之后的模型状态。你可以看到，前两个波到达并穿过第二个山峰，留下了一串空细胞。你还可以看到这个模型的动画版本，其中波形更清晰可见。

虽然这些波动由智能体组成，但我们可以将他们视为自己的实体，就像我们在“生命游戏”中想到的滑翔机一样。

这些波动的一个有趣的属性是，它们沿对角线移动，这可能是令人惊讶的，因为这些智能体本身只是向北或向东移动，从不向东北方移动。像这样的结果 - 团队或“集合”拥有智能体没有的属性和行为 - 在基于智能体的模型中很常见。我们将在下一章看到更多的例子。

9.8 涌现

本章中的例子展示了复杂性科学中最重要的想法之一：涌现。 涌现性是系统的一个特征，由它的成分相互作用而产生，而不是它们的属性。

为了澄清什么是涌现，考虑它不是什么会有帮助。 例如，砖墙很硬，因为砖和砂浆很硬，所以这不是涌现。 再举一个例子，一些刚性结构是由柔性部件构成的，所以这看起来像是一种涌现。 但它至多是一种弱例，因为结构特性遵循已知的力学定律。

相反，我们在谢林模型中看到的隔离是一种涌现，因为它不是由种族主义智能体造成的。 即使智能体只是轻微排外，系统的结果与智能体的决策意图有很大不同。

糖域中的财富分配可能是涌现，但它是一个弱例，因为我们可以根据视力，代谢和寿命的分布合理预测它。 我们在最后一个例子中看到的波动行为可能是一个更强的例子，因为波动显示出智能体显然没有的能力 - 对角线运动。

涌现性令人惊讶：即使我们知道所有规则，也很难预测系统的行为。难度不是偶然的；事实上，它可能是涌现的决定性特征。

正如沃尔夫勒姆在“新科学”中所讨论的那样，传统科学是基于这样的公理：如果你知道管理系统的规则，那么你可以预测它的行为。 我们所谓的“法律”通常是计算的捷径，它使我们能够预测系统的结果而不用建立或观察它。

但是许多细胞自动机在计算上是不可减少的，这意味着没有捷径。 获得结果的唯一方法是实现该系统。

一般而言，复杂系统可能也是如此。 对于具有多个成分的物理系统，通常没有产生解析解的模型。 数值方法提供了一种计算捷径，但仍存在质的差异。

解析解通常提供用于预测的恒定时间算法；也就是说，计算的运行时间不取决于预测的时间尺度`t`。 但数值方法，模拟，模拟计算和类似方法需要的时间与`t`成正比。 对于许多系统来说，我们无法计算出可靠的预测。

这些观察表明，涌现性基本上是不可预测的，对于复杂系统我们不应该期望，通过计算捷径来找到自然规律。

对某些人来说，“涌现”是无知的另一个名字; 根据这种思维，如果我们针对它没有还原论的解释，那么这个属性就是涌现的，但如果我们在将来更好地理解它，它就不再是涌现的。

涌现性的状况是有争议的话题，所以对此持怀疑态度是恰当的。 当我们看到明显的涌现性时，我们不应该假设永远不会有还原论解释。但我们也不应该假设必须有。

本书中的例子和计算等价原理提供了很好的理由，认为至少有些涌现性永远不会被古典还原论模型“解释”。

你可以在这里深入了解涌现：http://en.wikipedia.org/wiki/Emergence。

9.9 练习

练习 1

本章的代码位于本书仓库的 Jupyter 笔记本 `chap09.ipynb` 中。打开这个笔记本，阅读代码，然后运行单元格。你可以使用这个笔记本来练习本章的练习。我的解决方案在 `chap09soln.ipynb` 中。

练习 2

《The Big Sort》的作者 Bill Bishop 认为，美国社会越来越由政见所隔离，因为人们选择生活在志趣相投的邻居之中。

Bishop 所假设的机制并不是像谢林模型中的智能体那样，如果他们是孤立的，他们更有可能移动，但是当他们出于任何原因移动时，他们可能会选择一个社区，其中的人与他们自己一样。

修改谢林模型的实现来模拟这种行为，看看它是否会产生类似程度的隔离。

有几种方法可以模拟 Bishop 的假设。在我的实现中，随机选择的智能体会在每个步骤中移动。每个智能体考虑 `k` 个随机选择的空位置，并选择相似邻居的比例最高的位置。隔离程度和 `k` 有什么关系？

练习 3

在糖域的第一个版本中，我们从不添加智能体，所以一旦人口下降，它就不会恢复。 在第二个版本中，我们只是在智能体死亡时才取代，所以人口是不变的。 现在让我们看看如果我们增加一些“人口压力”会发生什么。

编写糖域的一个版本，在每一步结束时添加一个新的智能体。 添加代码来计算每个步骤结束时，智能体的平均视力和平均消耗。 运行模型几百步，绘制人口，平均视力和平均消耗随时间的变化。

你应该能够通过继承 `SugarScape` 并覆盖 `__init__` 和 `step` 来实现这个模型。

练习 4

在心灵哲学中，强人工智能是这样的理论，即受到适当编程的计算机可以拥有思想，与人类拥有的思想相同。

约翰·塞尔（John Searle）提出了一个名为“中文房间”的思想实验，旨在表明强 AI 是虚假的。你可以在 http://en.wikipedia.org/wiki/Chinese_room 上阅读。

对中文房间的论述的系统回复是什么？ 你对涌现的认识如何影响你对系统回复的反应？

十、兽群、鸟群和交通堵塞

> 原文：Chapter 10 Herds, Flocks, and Traffic Jams
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

本章的代码位于 `chap10.ipynb` 中，它是本书仓库中的 Jupyter 笔记本。使用此代码的更多信息，请参见第？节。

10.1 交通堵塞

是什么导致交通堵塞？在某些情况下，有明显的原因，如事故，车速监视或其他干扰交通的事情。 但其他时候，交通堵塞似乎没有明显的原因。

基于智能体的模型有助于解释自发性交通拥堵。 例如，我根据 Resnick，海龟，白蚁和交通堵塞模型实现了一个简单的高速路模拟。

这是代表“高速路”的类：

```
class Highway:

    def __init__(self, n=10, length=1000, eps=0):
        self.length = length
        self.eps = eps

        # create the drivers
        locs = np.linspace(0, length, n, endpoint=False)
        self.drivers = [Driver(loc) for loc in locs]

        # and link them up
        for i in range(n):
            j = (i+1) % n
            self.drivers[i].next = self.drivers[j]
```

`n` 是汽车的数量。

`length` 是高速路的长度，默认为 1000（以任意单位）。

`eps` 是我们将添加到系统中的随机噪声。

`loc` 包含驾驶员的位置；最初它们沿着高速公路等距分布。

驾驶员由 `Driver` 对象表示。 每个驾驶员都包含前方驾驶员的引用。 高速公路是圆形的，所以最后的驾驶员可以引用第一个。

`step` 方法简单；它只是移动每个驾驶员：

```
def step(self):
    for driver in self.drivers:
        self.move(driver)
```

这里是 `move` 方法：

```
def move(self, driver):
    d = self.distance(driver)

    # let the driver choose acceleration
    acc = driver.choose_acceleration(d)
    acc = min(acc, self.max_acc)
    acc = max(acc, self.min_acc)
    speed = driver.speed + acc

    # add random noise to speed
    speed *= np.random.uniform(1-self.eps, 1+self.eps)

    # keep it nonnegative and under the speed limit
    speed = max(speed, 0)
    speed = min(speed, self.speed_limit)

    # if current speed would collide, stop
    if speed > d:
        speed = 0

    # update speed and loc
    driver.speed = speed
    driver.loc += speed
```

`d` 是驾驶员与前方驾驶员之间的距离。 这个距离被传递给 `choose_acceleration` ，它规定了驾驶员的行为。 这是司机做出的唯一决定; 其他一切都由模拟的“物理”决定。

- `acc` 是加速度，它受 `min_acc` 和 `max_acc` 限制。 在我的实现中，汽车可以在 `max_acc = 1` 时加速，在 `min_acc = -10` 时加速。
- `speed` 是旧的速度加上请求的加速度，但是我们做了一些调整。 首先，我们向速度添加了随机噪音，因为这个世界并不完美。 `eps` 决定了噪音的幅度，这是适用于速度的百分比; 例如，如果 `eps` 为 0.02，则速度乘以 98% 到 102% 之间的随机数。
- 然后速度限制在 0 到 `speed_limit` 之间，在我的实现中为 40，所以汽车不允许后退或加速。
- 如果请求的速度会引起与下一辆车的碰撞，则速度设置为 0。
- 最后，我们更新驾驶员的速度和 `loc` 属性。

以下是 `Driver` 类的定义：

```
class Driver:

    def __init__(self, loc, speed=0):
        self.loc = loc
        self.speed = speed

    def choose_acceleration(self, d):
        return 1
```

`loc` 和 `speed` 属性是驾驶员的位置和速度。

`choose_acceleration` 的这个实现非常简单：它总是以最大速率加速。

由于汽车起步距离相等，因此我们预计它们都会加速，直到达到限速，或者直到它们的速度超过它们之间的距离。此时，至少会发生一次“碰撞”，导致一些汽车停下来。

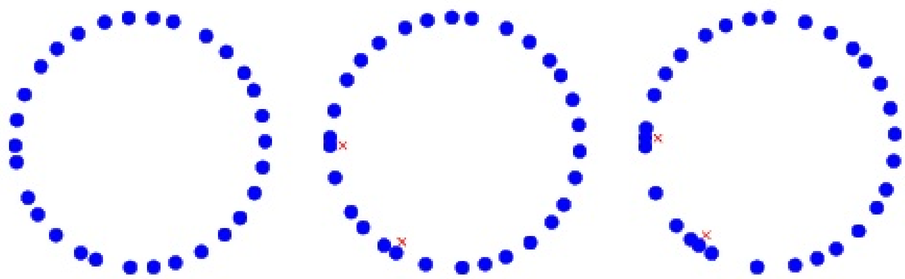


图 10.1：三个时间点中，环形公路上的驾驶员的模拟。点表示驾驶员的位置；十字表示驾驶员必须刹车来避开另一个驾驶员。

图？展示了该过程中的几个步骤，从 30 辆汽车和 `eps = 0.02` 开始。左边是 16 个时间步后的状态，汽车排列成一圈。由于随机噪音，有些汽车比其他汽车要快，并且间距变得不均匀。

在下一个时间步骤（中），两辆车相撞，用 `x` 标记表示。

在下一个时间步骤（右），两辆汽车会与已停车的汽车碰撞，我们可以看到最初形成的交通堵塞。一旦堵塞形成，它就会持续下去，其它汽车从后面靠近并碰撞，而前面的汽车加速离开。

在某些情况下，堵塞本身会向后传播，如果你观看本章的笔记本中的动画，你可以看到它。

10.2 随机噪声

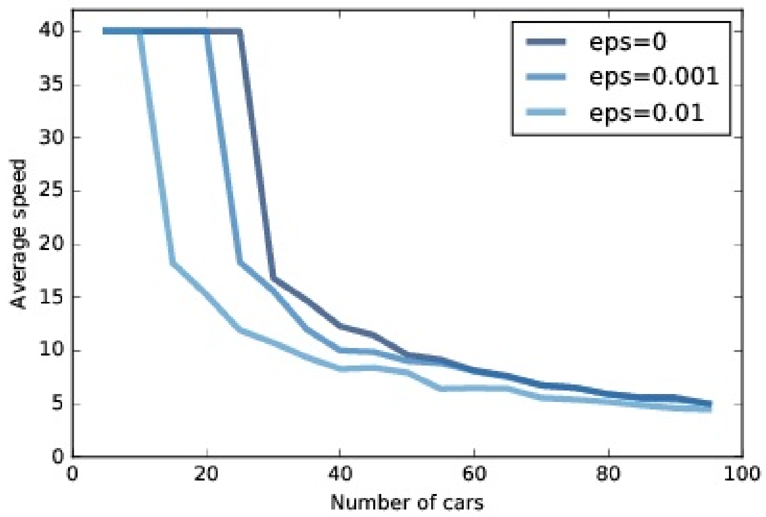


图 10.2：平均速度和汽车数量的函数，带有三个大小的附加随机噪声

随着汽车数量的增加，交通堵塞变得更加严重。 图？显示了汽车能够达到的平均速度，相对于汽车数量的函数。

最上面那行显示 `eps = 0` 的结果；也就是说，速度没有随机变化。 如果汽车数量少于 25 辆，则汽车之间的间隔大于 40，这样汽车可以达到并保持 40 的最大速度。超过 25 辆汽车形成交通堵塞，平均速度迅速下降。

这种效果是仿真物理学的直接结果，所以它不应该令人惊讶。 如果道路的长度为 1000，则 `n` 个车辆之间的间距为 `1000 / n` 。 而且由于汽车的行驶速度不超过前面的空间，所以我们预计，最高平均车速为 `1000 / n` 或 40，取最小者。

但这是最好的情况。只有少量的随机性，情况会变得更糟。

图？也显示了 `eps = 0.001` 和 `eps = 0.01` 的结果，其对应于 0.1% 和 1% 的速度误差。

即使有少量噪音，高速路的容量也会从 25 降至 20（“容量”是指可以达到并保持速度限制的最大车辆数量。 如果有 1% 的误差，容量会下降到 10。

作为本章结尾的练习之一，你将有机会设计出更好的驾驶员; 也就是说，你将在 `choose_acceleration` 中尝试不同的策略，并查看你是否可以找到可提高平均速度的驾驶行为。

10.3 Boids

1987 年，Craig Reynolds 发表了《兽群，鸟群和鱼群：分布式行为模型》（Flocks, herds and schools: A distributed behavioral model），描述了一个基于智能体的兽群行为模型。 你可以从 http://www.red3d.com/cwr/papers/1987/boids.html 下载他的论文。

这种模型中的智能体被称为“boids”，既是“bird-oid”的缩写，又是“bird”的口音发音（虽然 boids 也用于模拟鱼类和集中的陆生动物）。

每个智能体模拟了三种行为：

避免碰撞：

避开障碍物，包括其他鸟类。

鸟群集中：

移向鸟群的中心。

速度匹配：

将速度（速率和方向）与邻近的鸟类对齐。

Boid 只根据局部信息做出决定；每个 boid 只能看到（或注意）其视野范围内的其他 boid。

在本书的仓库中，你会发现 `Boids7.py`，它包含我的 boids 实现，部分基于《Flake, The Computational Beauty of Nature》（雪花：自然的计算之美）中的描述。

该程序定义了两个类：`Boid`，实现了 boid 算法，和 `World`，包含 `Boid` 列表和吸引 `Boid` 的“胡萝卜”列表。

boid 算法使用 `get_neighbors` 在视野中查找其他 boid：

```
def get_neighbors(self, others, radius, angle):
    boids = []
    for other in others:
        if other is self:
           continue

        offset = other.pos - self.pos

        # if not in range, skip it
        if offset.mag > radius:
            continue

        # if not within viewing angle, skip it
        if self.vel.diff_angle(offset) > angle:
            continue

        # otherwise add it to the list
        boids.append(other)

    return boids
```

`get_neighbors` 使用向量减法来计算从 `self` 到 `other` 的向量。 这个向量的们是到另一个 boid 的距离。 `diff_angle` 计算 `self` 的速度（也是视线）与另一个 boid 之间的角度。

`center` 寻找视野中 boid 的质心，并返回一个指向它的向量：

```
def center(self, others):
    close = self.get_neighbors(others, r_center, a_center)
    t = [other.pos for other in close]
    if t:
        center = sum(t)/len(t)
        toward = vector(center - self.pos)
        return limit_vector(toward)
    else:
        return null_vector
```

同样，`avoid`寻找范围内任何障碍物的质心，并返回一个指向它的向量，`copy`将返回当前朝向和邻居的平均朝向之间的差，`love`计算出胡萝卜的朝向。

`set_goal`计算这些目标的加权总和并设定总体目标：

```
def set_goal(self, boids, carrot):
    self.goal = (w_avoid * self.avoid(boids, carrot) +
                 w_center * self.center(boids) +
                 w_copy * self.copy(boids) +
                 w_love * self.love(carrot))
```

最后`move`更新 boid 的速度，位置和姿势。

```
def move(self, mu=0.1):
    self.vel = (1-mu) * self.vel + mu * self.goal
    self.vel.mag = 1

    self.pos += dt * self.vel
    self.axis = b_length * self.vel.norm()
```

新速度是旧速度和目标的加权和。 参数`mu`决定鸟类能够多快地改变速度和方向。 时间步长`dt`决定了 boids 移动的距离。

许多参数影响鸟群行为，包括每个行为的范围，角度和权重以及可操作性`mu`。

这些参数决定了 boids 形成和维持鸟群的能力，以及鸟群中运动和组织的模式。 对于某些设置，boids 类似于一群鸟；其他设置类似于鱼群或一片飞虫。

10.4 涌现和自由意志

作为一个整体，许多复杂的系统具有它们的组件不具有的属性：

- 细胞自动机规则 30 是确定性的，控制其演化的规则是完全已知的。 尽管如此，它会生成一个序列，统计上与随机无法区分。
- 谢林模型中的智能体不是种族主义者，但他们互动的结果就好像他们是。
- 糖域中的智能体形成对角移动的波浪，尽管智能体不能。
- 即使汽车正在向前行驶，交通堵塞会向后移动。
- 兽群和鸟群的行为来自其成员之间的局部互动。

这些例子提出了一个途径，用于解决几个古老而富有挑战性的问题，包括意识和自由意志的问题。

自由意志是做出选择的能力，但是如果我们的身体和大脑受到确定性物理规律的支配，我们的选择就会是确定的。 自由意志的争论无数；我只会提到两个：

- 威廉·詹姆斯（William James）提出了一个两阶段模型，其中可能的行为由随机过程产生，然后由确定性过程选择。 在这种情况下，我们的行为基本上是不可预测的，因为生成它们的过程包含随机元素。
- 大卫休谟（David Hume）认为，我们对于做出选择的感知是一种幻觉；在这种情况下，我们的行为是确定性的，因为产生它们的系统是确定性的。

这些论点以相反的方式调解冲突，但他们同意冲突是存在的：如果这些部分是确定性的，那么系统就不会有自由意志。

本书中的复杂系统提出了另一种选择，在选择和决策层面的自由意志，相当于神经元层面的（或更低层次）的决定论。 就像汽车向前行驶时，交通堵塞后退的方式一样，即使神经元没有，人也可以有自由意志。

10.5 练习

练习 1

在交通堵塞的模拟中，定义一个类，`BetterDriver`，它继承`Driver`并覆盖`choose_acceleration`。 查看你是否可以定义一个驾驶规则，比`Driver`中的基本实现更好的。 你可能会尝试到达更高的平均速度，或者更少的碰撞。

练习 2

注意：为了做这个练习，你必须安装 VPython，一个用于 3D 显示和动画的库。 如果你使用 Anaconda（我在第？节中推荐过），你可以执行：

```
conda install -c vpython vpython
```

然后运行本书仓库中的`Boids7.py`。 阅读代码来查看，程序开始时定义的参数如何控制 boid 的行为。 试验不同的参数。 如果通过将权重设置为 0 来“关闭”其中一种行为，会发生什么？

为了生成更多类似鸟类的行为，Flake 建议增加第四种行为来保持清晰的视线；换句话说，如果在正前方有另一只鸟，那么它就应该从侧面移开。 你认为这个规则对鸟群的行为有什么影响？ 实现它来看看。

练习 3

在 http://en.wikipedia.org/wiki/Free_will 上深入了解自由意志。自由意志与决定论相容的观点被称为相容论。相容论最大的挑战之一是“结果论证”（consequence argument）。什么是结果论证？根据你在本书中读到的内容，你对结果论证有什么样的反应？

十一、进化

> 原文：Chapter 11 Evolution
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

生物学乃至整个科学最重要的思想，是通过自然选择的进化论，它声称由于自然选择而创造出新的物种并改变现有的物种。自然选择是个体间遗传差异导致生存和繁殖差异的过程。

在了解生物学的人中，进化论被广泛认为是一个事实，也就是它足以接近事实，如果将来得到纠正，纠正将使中心思想基本保持完整。

尽管如此，许多人并不相信进化论。在皮尤研究中心进行的一项调查中，被调查者被问到，以下哪些断言更贴近他们的观点：

- 人类和其他生物随时间而进化。
- 起初，人类和其他生物就以其现在的形式存在。

约有 34% 的美国人选择了第二个（见 http://www.thearda.com/Archive/Files/Codebooks/RELLAND14_CB.asp）。

即使在那些认为生物已经进化的人中，只有一半以上的人认为进化的原因是自然选择。 换句话说，只有三分之一的美国人相信进化论是真实的。

这怎么可能？ 在我看来，促成因素包括：

- 有些人认为进化论与他们的宗教信仰之间有冲突。 感觉就像他们不得不拒绝一个，他们拒绝了进化论。
- 其他人经常被第一组成员积极误导，以至于他们对进化论的许多认识都是错误的。
- 许多人根本就不了解进化。

对于第一组，我可能没有太多可以做的事，但我认为我可以帮助其他人。 经验上，进化论很难让人理解。 同时，它非常简单：对很多人来说，一旦他们了解进化论，它似乎既明显又无可辩驳。

为了帮助人们从困惑转变为清晰，我找到的最强大的工具就是计算。 我们看到，理论上很难理解的想法，在模拟中出现时很容易理解。 这是本章的目标。

本章的代码位于 `chap11.ipynb` 中，该书是本书仓库中的 Jupyter 笔记本。使用此代码的更多信息，请参见第？节。

11.1 简单的进化

我将从一个简单的模型开始，演示一种基本的进化形式。 根据该理论，以下特征足以产生进化：

- 复制者：我们需要一批能够以某种方式复制的智能体。 我们将以复制者开始，它们生成它们自己的完美的副本。 稍后我们将添加不完美的副本，即突变。
- 突变：我们还需要一些种群中的变化，也就是个体之间的差异。
- 生存和繁殖差异：个体之间的差异必须影响其生存或繁殖的能力。

为了模拟这些特征，我们将定义智能体种群，智能体代表个体。 每个智能体都有遗传信息，称为基因型，这是智能体繁殖时复制的信息。 在我们的模型中 [1]，基因型由`N`个二进制数字（零和一）的序列表示，其中`N`是我们选择的参数。

> [1] 模型是主要由 Stuart Kauffman 开发的 NK 模型的变体（参见 https://en.wikipedia.org/wiki/NK_model）。

为了产生突变，我们创建了具有多种基因型的种群；稍后我们将探讨创造或增加突变的机制。

最后，为了产生生存和繁殖差异，我们定义了一个函数，将每个基因型映射为一个适应度，其中适应度是一个数量，有关智能体的生存或繁殖能力。

11.2 适应性景观

将基因型映射为适应性函数，称为适应性景观。 在景观的隐喻中，每个基因型对应于`N`维空间中的一个位置，并且适应性对应于该位置处的景观的“高度”。对于能够解释这个隐喻的可视化，参见 https://en.wikipedia.org/wiki/Fitness_landscape。

在生物学术语中，适应性景观代表一种信息，它是生物体的基因型与其物理形式和能力的关系，后者称为其表现型，以及表现型如何与其环境相互作用。

在现实世界中，适应性景观很复杂，但我们不需要建立现实模型。 为了诱导进化，我们需要基因型和适应性之间的某种关系，但事实证明它可以是任何关系。 为了证明它，我们将使用完全随机的适应性景观。

这是代表适应性景观的类的定义：

```
class FitnessLandscape:
    def __init__(self, N):
        self.N = N
        self.one_values = np.random.random(N)
        self.zero_values = np.random.random(N)

    def fitness(self, loc):
        fs = np.where(loc, self.one_values,
                           self.zero_values)
        return fs.mean()
```

智能体的基因型，对应其在适应性景观中的位置，由一个 NumPy 的零一数组来表示，称为 `loc`。给定基因型的适应性，是 `N` 个适应性贡献的平均值，`loc` 的每个元素都是一个。

为了计算基因型的适应性，`FitnessLandscape` 使用两个数组：`one_values`，其中包含 `loc` 的每个元素都为 1 时的适应性贡献，以及 `zero_values`，其中包含为 0 时的适应度贡献。

`fitness` 方法使用 `np.where`，如果 `loc` 中的值为 1，它从 `one_values` 中选择一个值，如果 `loc` 中的值为 0，它从 `zero_values` 中选择一个值。

例如，假设 `N=3`：

```
one_values =  [0.1, 0.2, 0.3]
zero_values = [0.4, 0.7, 0.9]
```

这种情况下，`loc = [0, 1, 0]` 的适应性是 `[0.4, 0.2, 0.9]` 的均值，为 0.5。

11.3 智能体

接下来我们需要智能体，这是类定义：

```
class Agent:

    def __init__(self, loc, fit_land):
        self.loc = loc
        self.fit_land = fit_land
        self.fitness = fit_land.fitness(self.loc)

    def copy(self):
        return Agent(self.loc, self.fit_land)
```

智能体的属性是：

`loc`：智能体在适应性景观中的位置。`fit_land`：`FitnessLandscape` 对象的引用。`fitness`：智能体在 `FitnessLandscape` 中的适应性，表示为 0 到 1 之间的数字。`Agent` 的这个定义提供了一种简单的 `copy` 方法，可以精确复制基因型；之后，我们将看到一个带有突变的版本，但突变对于进化来说不是必需的。

11.4 模拟

现在我们有了智能体和适应性景观，我将定义一个名为 `Simulation` 的类，用于模拟智能体的创建，繁殖和死亡。 为了避免陷入困境，我将在这里提供一个简化版本的代码；你可以在本章的笔记本上看到细节。

这是 `Simulation` 的定义：

```
class Simulation:

    def __init__(self, fit_land, agents):
        self.fit_land = fit_land
        self.agents = agents
```

`Simulation` 的属性是：

- `fit_land`：`FitnessLandscape` 对象的引用。
- `agents`：`Agent` 对象的数组。

`Simulation` 中最重要的函数是 `step`，它模拟了单个时间步骤：

```
# class Simulation:

    def step(self):
        n = len(self.agents)
        fits = self.get_fitnesses()

        # see who dies
        index_dead = self.choose_dead(fits)
        num_dead = len(index_dead)

        # replace the dead with copies of the living
        replacements = self.choose_replacements(num_dead, fits)
        self.agents[index_dead] = replacements
```

在每个时间步骤中，一些智能体死亡，一些智能体繁殖。 `step` 使用另外三个方法：

- `get_fitnesses` 返回一个数组，包含每个智能体的适应性，按照它们在智能体数组中出现的顺序。
- `choose_dead` 决定哪些智能体在此时间步中死亡，并返回一个数组，包含死亡智能体的索引。
- `choose_replacements` 决定哪些智能体在此时间步中繁殖，在每个智能体上调用 `copy`，并返回一个新的 `Agent` 对象的数组。

在这个版本的模拟中，每个时间步中新智能体的数量等于死亡智能体的数量，所以活动智能体的数量是恒定的。

11.5 没有差异

在我们运行模拟之前，我们必须指定`choose_dead`和`choose_replacements`的行为。 我们将从这些函数的简单版本开始，它们不依赖于适应性：

```
# class Simulation

     def choose_dead(self, fits):
        n = len(self.agents)
        is_dead = np.random.random(n) < 0.1
        index_dead = np.nonzero(is_dead)[0]
        return index_dead

    def choose_replacements(self, n, fits):
        agents = np.random.choice(self.agents, size=n, replace=True)
        replacements = [agent.copy() for agent in agents]
        return replacements
```

在`choose_dead`中，`n`是智能体的数量，`is_dead`是一个布尔数组，对于此时间步骤内死亡的智能体为`True`。 在这个版本中，每个智能体都有相同的死亡概率：0.1。 `choose_dead`使用`np.nonzero`来查找`is_dead`的非零元素的索引（`True`被视为非零）。

在`choose_replacements`中，`n`是在此时间步骤中复制的智能体数量。 它使用`np.random.choice`带替换地选择`n`个智能体。 然后它在每个上调用`copy`，并返回一个新的`Agent`对象列表。

这些方法不依赖于适应性，所以这种模拟没有生存或繁殖差异。 因此，我们不应期待看到进化。 但是，我们怎么辨别呢？

11.6 进化的证据

进化的最具包容性的定义是，种群中基因型分布的变化。 进化是一种聚合效应：换句话说，个体不会进化；但种群会。

在这个模拟中，基因型是高维空间中的位置，因此很难将其分布中的变化可视化。 但是，如果基因型改变，我们预计它们的适应性也会改变。 所以我们将将适应性分布的变化用作进化的证据。 具体来说，我们将看看种群中适应性的均值和标准差。

在我们运行模拟之前，我们必须添加一个`Instrument`，它是在每个时间步骤后更新的对象，计算一个感兴趣的统计量，并将结果存储在一个序列中，我们稍后可以绘制它。

这是所有仪器的父类：

```
class Instrument:
    def __init__(self):
        self.metrics = []
```

下面是`MeanFitness`的定义，`MeanFitness`是一个仪器，计算每个时间步的种群平均适应性：

```
class MeanFitness(Instrument):
    def update(self, sim):
        mean = np.nanmean(sim.get_fitnesses())
        self.metrics.append(mean)
```

现在我们准备好运行模拟了。 为了最小化起始种群中随机变化的影响，我们使用同一组智能体启动每个模拟。 为了确保我们探索整个适应性景观，我们由每个位置的一个智能体开始。以下是创建模拟的代码：

```
N = 8
fit_land = FitnessLandscape(N)
agents = make_all_agents(fit_land, Agent)
sim = Simulation(fit_land, agents)
```

`make_all_agents` 为每个位置创建一个智能体； 本章的实现在笔记本中。

现在我们可以创建并添加 `MeanFitness` 仪器，运行模拟并绘制结果：

```
instrument = MeanFitness()
sim.add_instrument(instrument)
sim.run()
sim.plot(0)
```

模拟维护了 `Instrument` 对象列表。 在每个时间步之后，它在列表中的每个仪器上调用 `update` 。

模拟运行后，我们使用 `Simulation.plot` 绘制结果，它接受索引作为参数，使用索引从列表中选择一个 `Instrument` 并绘制结果。 在这个例子中，只有一个 `Instrument` ，索引为 0。

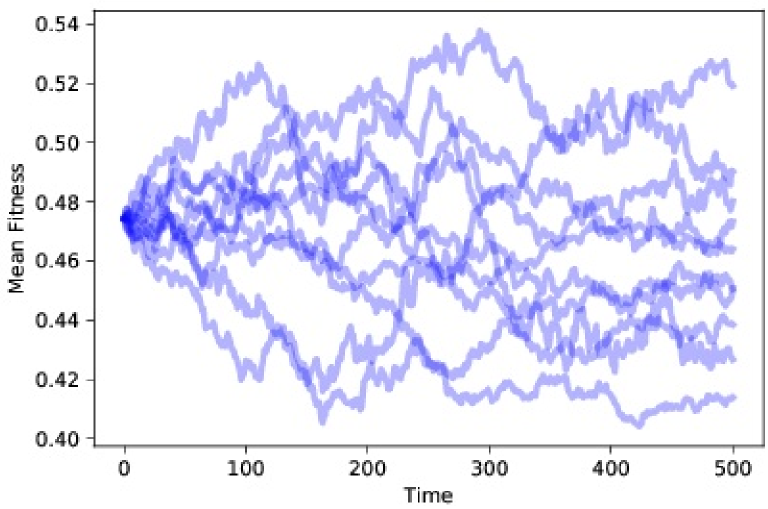


图 11.1：随着时间的推移，10 次模拟的平均适应性，没有生存或繁殖差异

图？显示了运行这个模拟 10 次的结果。 种群的平均适应性随机移动。 由于适应性的分布随时间变化，我们推断表现型的分布也在变化。 按照最具包容性的定义，这种随机游走是一种进化。 但它不是一个特别有趣的类型。

特别是，这种进化并不能解释生物物种如何随时间变化，或者如何出现新的物种。 进化论是强大的，因为它解释了我们在自然界看到的似乎无法解释的现象：

- 适应性：物种与其环境的相互作用似乎太复杂，太巧妙，并且偶然发生。 自然系统的许多特征看起来好像是设计出来的。
- 增加的多样性：地球上的物种数量随时间而普遍增加（尽管有几个时期的大规模灭绝）。
- 增加的复杂性：地球上的生命史起始于相对简单的生命形式，后来在地质记录中出现了更复杂的生物体。

这些是我们想要解释的现象。 到目前为止，我们的模型并没有完成这个任务。

11.7 生存差异

让我们再添加一种成分，生存差异。 以下是继承 `Simulation` 并覆盖 `choose_dead` 的类的定义：

```
class SimWithDiffSurvival(Simulation):

    def choose_dead(self, fits):
        n = len(self.agents)
        is_dead = np.random.random(n) > fits
        index_dead = np.nonzero(is_dead)[0]
        return index_dead
```

现在生存的概率取决于适应性；事实上，在这个版本中，智能体在每个时间步骤中幸存的概率是其适应性。

由于适应性低的智能体更有可能死亡，因此适应性高的智能体更有可能生存足够长的时间来繁殖。 我们预计适应性低的智能体的数量会随时间而减少，适应性高的智能体的数量会增加。

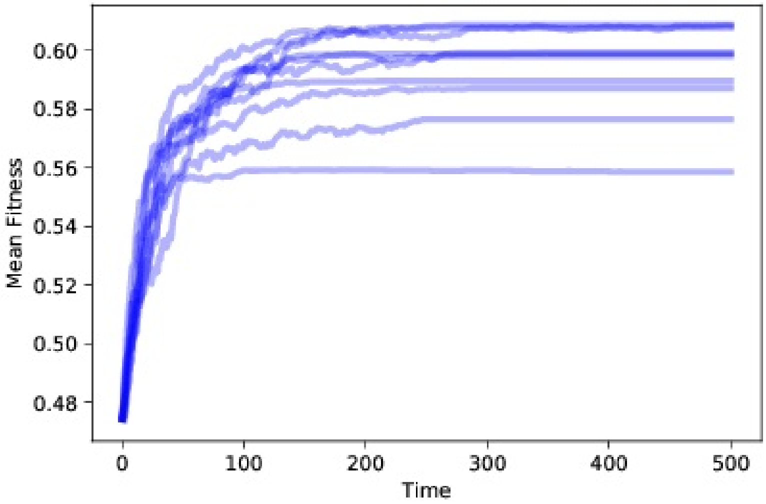


图 11.2：随着时间的推移，10 次模拟中的适应性均值，带有生存差异

图？显示了随着时间的推移，10 次模拟中的适应性均值，带有生存差异。 平均适应性起初会迅速增加，但会逐渐平稳。

你或许可以弄清楚为什么它会平稳：如果在特定位置只有一个智能体并且它死了，它就会使这个位置变空。没有突变，就没有办法让它再次被占领。

在 `N = 8` 的情况下，该模拟以 256 个智能体开始，它们占用了所有可能位置。 占用位置的数量随时间而减少；如果模拟运行时间足够长，最终所有智能体将占用相同的位置。

所以这个模拟开始解释适应性：增加的适应性意味着，物种在它的环境中生存得更好。 但是占用位置的数量随时间而减少，所以这个模型根本无法解释增加的多样性。

在本章的笔记本中，你将看到差异化繁殖的效果。 正如你所预料的那样，差异化繁殖也会增加平均适应性。但没有突变，我们仍然没有看到增加的多样性。

11.8 突变

在目前的模拟中，我们以可能的最大多样性开始 - 在景观的每个位置都有一个智能体 - 并以可能的最小多样性结束，所有智能体都在一个位置。

这与自然界发生的情况几乎相反，它显然以单个物种开始，这种物种随时间而分化为今天的地球上数百万甚至数十亿物种（见 https://en.wikipedia.org/wiki/Global_biodiversity）。

使用我们模型的完美复制，我们从未看到增加的多样性。 但是如果我们加上突变，再加上生存和繁殖差异，我们距离理解自然界的进化就更近了一步。

以下是继承 `Agent` 并覆盖 `copy` 的类定义：

```
class Mutant(Agent):

    prob_mutate = 0.05

    def copy(self):
        if np.random.random() > self.prob_mutate:
            loc = self.loc.copy()
        else:
            direction = np.random.randint(self.fit_land.N)
            loc = self.mutate(direction)
        return Mutant(loc, self.fit_land)
```

在这种突变模型中，每次我们调用 `copy` 时，都有 5% 的突变机会。 在突变的情况下，我们从当前位置选择一个随机方向 - 即基因型中的一个随机位 - 并翻转它。 这是 `mutate` ：

```
    def mutate(self, direction):
        new_loc = self.loc.copy()
        new_loc[direction] ^= 1
        return new_loc
```

运算符 `^=` 计算“异或”；操作数 1 具有翻转一位的效果（请参阅 https://en.wikipedia.org/wiki/Exclusive_or#Bitwise_operation）。

现在我们有了突变，我们不必在每个位置都放置一个智能体。 相反，我们可以以最小变化开始：所有智能体在同一位置。

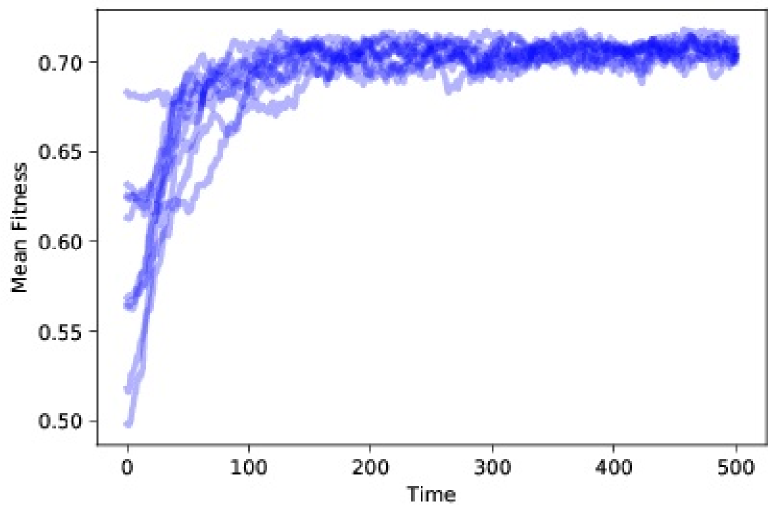


图 11.3：随着时间的推移，10 次模拟中的适应性均值，带有突变、生存繁殖差异

图？显示了 10 次模拟的结果，带有突变和生存繁殖差异。 在任何情况下，种群都会向最大适应性的位置进化。

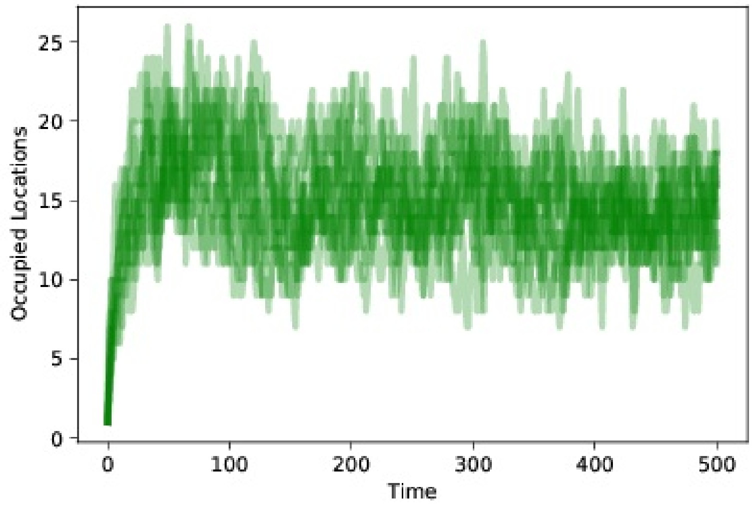


图 11.4：随着时间的推移，10 次模拟的占用位置的数量，带有突变和生存繁殖差异。

为了测量种群的多样性，我们可以绘制每个时间步后占用位置的数量。 图？展示了结果。 我们以同一地点的 100 个智能体开始。 随着突变的发生，占用位置的数量迅速增加。

当智能体发现适应性高的位置时，它更有可能生存和繁殖。 适应性较低的位置上的智能体最终消失。 种群在整个景观中随时间而移动，直到大多数智能体处于适合性最高的位置。

此时，系统达到平衡，突变以相同的速率占据新的位置，生存差异导致适合性低的位置清空。

平衡中的占用位置的数量，取决于突变率和生存差异的程度。 在这些模拟中，任何点处的独特占用位置的数量通常为 10-20。

重要的是要记住，这个模型中的智能体不会移动，就像生物体的基因型没有改变一样。 当智能体死亡时，它可能会留下一个空位。 当发生突变时，它可以占据一个新的位置。 当智能体从某些地方消失并出现在其他地方时，种群会在景观中移动，就像生命游戏中的滑翔机一样。 但生命体不会进化；但种群可以。

11.9 物种形成

进化论说，自然选择改变了现有的物种并创造了新的物种。 在我们的模型中，我们看到了变化，但我们并没有看到新的物种。 在模型中，还不清楚新物种是什么样。

在有性繁殖的物种中，如果两种生物能够繁殖并产生丰富的后代，则被视为同一物种。 但是模型中的智能体不会再现性行为，所以这个定义不适用。

在无性繁殖的生物中，如细菌，物种的定义并不明确。 一般来说，如果一个种群的基因型形成一个簇，那么它就被认为是一个物种，也就是说，如果种群内的遗传差异比种群间的差异小。

在我们可以对新物种建模之前，我们需要能够识别景观中的智能体簇，这意味着我们需要定义位置之间的距离。 由于位置是用二进制数字串表示的，因此我们将距离定义为基因型中不同的位数。 `FitnessLandscape` 提供了 `distance` 方法：

```
# class FitnessLandscape

    def distance(self, loc1, loc2):
        return np.sum(np.logical_xor(loc1, loc2))
```

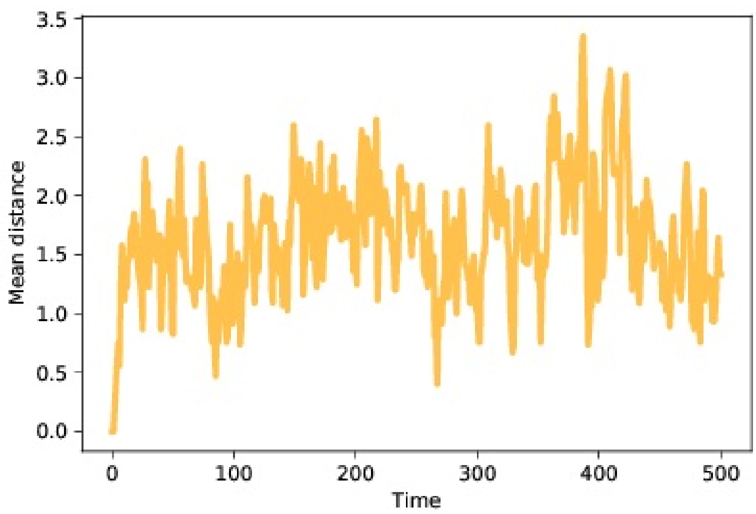


图 11.5：智能体随时间变化的平均距离

`logical_xor` 函数计算“异或”，不同的元素为 `True` ，相同的元素为 `False` 。

为了量化种群的分散，我们可以计算每对智能体之间距离的平均值。 在本章的笔记本中，你会看到 `MeanDistance` 仪器，它会在每个时间步骤后计算这个度量。

图？ 展示了智能体随时间的平均距离。 因为我们从相同的突变开始，所以初始距离为 0。随着突变的发生，平均距离增加，在种群遍布景观时达到最大值。

一旦智能体发现最佳位置，平均距离就会减小，直到种群达到平衡，由于突变引起的距离增加通过距离的减小而平衡，因为远离最佳位置的智能体更有可能死亡。 在这些模拟中，平衡时的平均距离接近 1.5；也就是说，大多数智能体距离最佳位置只有 1-2 个突变。

现在我们准备寻找新的物种。 为了模拟一种简单的物种形成，假设一个种群在不变的环境中演化，直到它达到稳定状态（就像我们在自然界发现的一些物种，似乎在很长一段时间内变化很小）。

现在假设我们改变环境，或者将种群转移到新的环境中。 一些旧环境中适应性较高的特性，可能会在新环境中适应性较低，反之亦然。

我们可以通过运行模拟来模拟这些情景，直到种群达到稳定状态，然后改变适应性景观，然后恢复模拟，直到种群再次达到稳定状态。

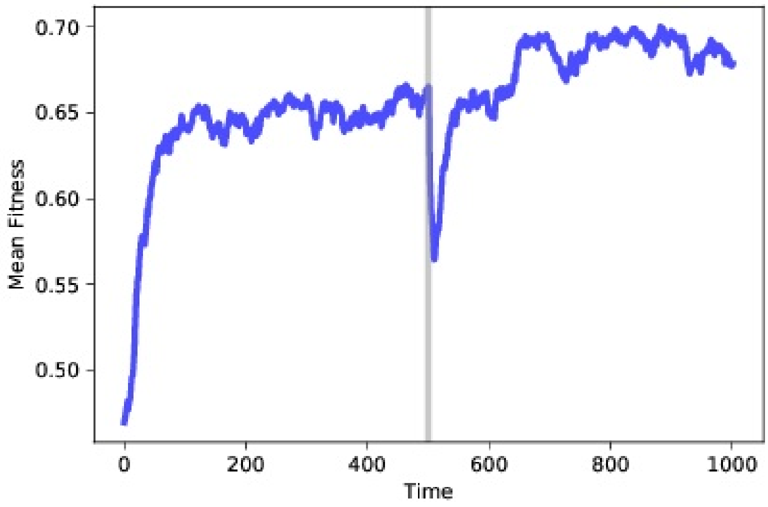


图 11.6：随时间变化的适应性均值。在 500 步之后，我们改变了适应性景观

图？展示了这样的模拟结果。 再次，我们从随机位置开始，使用 100 个相同的突变体，并运行 500 个时间步骤的模拟。 在这个时候，许多智能体处于最佳位置，在这个例子中，其适应性接近 0.65。 智能体的基因型形成一个簇，智能体之间的平均距离接近 1。

经过 500 步之后，我们运行 `FitnessLandscape.set_values` ，这改变了适应性景观； 然后我们恢复模拟。 平均适应性会立即下降，因为旧景观中的最佳位置并不比新景观中的随机位置好。

然而，当种群迁移到新景观时，平均适应性会迅速增加，最终会找到新的最佳值，其适应度接近 0.75（在这个例子中恰好更高，但不一定是）。

一旦种群达到稳定状态，它就会形成一个新的簇，智能体之间的平均距离再次接近 1。

现在，如果我们计算智能体之前和之后的位置之间的距离，平均而言，它们相差超过 6。 簇之间的距离远大于每个簇内的距离，因此我们可以将这些簇解释为不同的物种。

11.10 总结

我们已经看到，突变以及生存和繁殖差异，足以导致适应性的增加，多样性的增加，并产生简单形式的物种。 这种模型并不是现实的；自然系统中的进化要比这复杂得多。 相反，它意味着一个“充足定理”；也就是说，模型的特征足以产生我们试图解释的行为（参见 https://en.wikipedia.org/wiki/Necessity_and_sufficiency）。

从逻辑上讲，这个“定理”并不能证明，自然界中的进化仅仅由这些机制引起，但是由于这些机制确实以多种形式出现在生物系统中，所以认为它们至少有助于自然进化，是合理的。

同样，该模型并不能证明这些机制总是会导致进化。 但是我们在这里看到的结果是可靠的：在几乎所有包含这些特征的模型中 - 不完美的复制者，变异性和繁殖差异 - 发生了进化。

我希望这一观察有助于揭开进化的神秘面纱。 当我们观察自然系统时，进化看起来很复杂。而且由于我们主要看到了进化的结果，而没有看到这个过程，所以难以想象和相信。

但在模拟中，我们看到整个过程，而不仅仅是结果。 通过包含最少的一系列特征来产生进化 - 暂时忽略了生物生命的巨大复杂性 - 我们可以将进化看作是一个令人惊讶的简单，不可避免的想法。

11.11 练习

本章的代码位于本书仓库的 Jupyter 笔记本`chap11.ipynb`中。 打开笔记本，阅读代码并运行单元格。 笔记本包含本章的练习。 我的解决方案在`chap11soln.ipynb`中。

十二、合作进化

> 原文：Chapter 12 Evolution of cooperation
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

在最后一章中，我们提出两个问题，一个来自生物学，一个来自哲学：

- 在生物学中，“利他主义问题”是自然选择与利他主义之间的明显冲突，自然选择表明动物生存在不断竞争的状态中来生存和繁殖，利他主义是许多动物帮助其他动物的倾向，甚至是显然对他们不利。见 https://en.wikipedia.org/wiki/Altruism_(biology)。
- 在道德哲学中，人性问题是，人类是否从根本上是善良的，或者邪恶的，或者是由环境塑造的空白状态。见 https://en.wikipedia.org/wiki/Human_nature。

我们将用来解决这些问题的工具，（同样）是基于智能体的模拟和博弈论，博弈论是一组抽象模型，旨在描述智能体交互的各种方式。具体来说，我们会考虑囚徒困境。

本章的代码位于`chap12.ipynb`中，该书是本书仓库中的`Jupyter`笔记本。使用此代码的更多信息，请参见第？节。

12.1 囚徒困境

囚徒困境是博弈论中的一个话题，但它不是一种有趣的博弈。相反，这种博弈揭示了人类的动机和行为。以下是来自维基百科的它的介绍（http://en.wikipedia.org/wiki/Prisoner's_dilemma）：

两名犯罪团伙成员被逮捕并囚禁。每个囚犯都被单独监禁，无法与另一方交流。检察官缺乏足够的证据，来证明这两个人的主要指控。他们希望以较轻的指控被判处两年徒刑。同时，检察官为每个囚犯提供商量的余地。每个囚犯都有机会：（1）通过证明对方犯罪出卖对方，或（2）通过保持沉默与另一方合作。出价是：

- 如果 A 和 B 各自背叛对方，每个人都服刑 2 年。
- 如果 A 背叛 B 但 B 保持沉默，A 将被释放，B 将被监禁 3 年（反之亦然）。
- 如果 A 和 B 都保持沉默，他们两人只会服刑 1 年（较轻的质控）。

很显然，这种情况是假想的，但它用于代表各种不同的互动，其中智能体必须选择是相互“合作”还是“背叛”，以及每个智能体的奖励（或惩罚）取决于他人的选择。

有了这套奖惩，我们很有可能说智能体应该合作，也就是说，双方都应该保持沉默。 但两个智能体不知道对方会做什么，所以每个人都必须考虑两种可能的结果。 首先，从 A 的角度来看它：

- 如果 B 保持沉默，A 最好是背叛；他会无罪而不是服刑 1 年。
- 如果 B 背叛，A 最好也是背叛；他只会服刑 2 年而不是 3 年。

不管 B 做什么，A 最好都是背叛。 而且因为博弈是对称的，所以从 B 的角度来看这个分析是一样的：不管 A 做什么，B 最好也是背叛。

在这个博弈的最简单版本中，我们假设 A 和 B 没有考虑其他因素。 他们不能互相沟通，所以他们不能协商，作出承诺或相互威胁。 他们只考虑直接目标，最小化他们的判决；他们不考虑任何其他因素。

在这些假设下，两个智能体的理性选择都是背叛。 这可能是一件好事，至少在刑事司法方面是这样。 但对囚犯来说，这令人沮丧，因为显然，他们无法获得他们双方都想要的结果。 而且这种模式适用于现实生活中的其他场合，其中合作有更大的好处以及对于玩家来说都会更好。

研究这些场景以及摆脱困境的方法，博弈论研究者关注的焦点，但这不是本章的重点。 我们正朝着不同的方向前进。

12.2 善良的问题

自 20 世纪 50 年代，囚徒困境被首次讨论以来，它一直是社会心理学研究的热门话题。根据前一节的分析，我们可以说一个理想的智能体应该做什么; 很难预测真正的人究竟做了些什么。 幸运的是，实验已经完成了 [1]。

> [1] 这里有一个最近的报告，提到以前的实验：Barreda-Tarrazona, Jaramillo-Gutiérrez, Pavan, and Sabater-Grande, “Individual Characteristics vs. Experience: An Experimental Study on Cooperation in Prisoner’s Dilemma", Frontiers in Psychology, 2017; 8: 596. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5397528/。

如果我们假设人们足够聪明地进行分析（或者在解释时理解它），并且他们通常为了自己的利益而行事，那么我们预计他们几乎总是背叛。 但他们没有。 在大多数实验中，受试者的合作远远超过理性的智能体模型的预测 [2]。

> [2] 有个不错的视频归纳了我们目前讨论的内容：https://www.youtube.com/watch?v=t9Lo2fgxWHw。

这个结果最明显的解释是，人们不是理性的智能体，这对任何人都不应该感到惊讶。 但为什么不是呢？ 是因为他们不够聪明，无法理解这种情况，还是因为他们故意违背自己的利益行事？

根据实验结果，似乎至少有一部分解释是纯粹的利他主义：许多人愿意为了让别人受益而承担成本。现在，在你提出《Journal of Obvious Results》上发表的结论之前，让我们继续问为什么：

- 为什么人们会帮助别人，即使自己会付出代价？至少部分原因是他们想这样；这让他们对自己和世界感觉良好。
- 为什么善良让人感觉良好？诱人的说法是，有人跟他们提出这是正确的，或者更普遍来说，他们被社会训练为想要做好事。但毫无疑问 [3]，至少有一大部分利他主义是天生的；在不同程度上，利他主义的倾向是正常大脑发育的结果。
- 那么，为什么呢？大脑发育的内在部分，以及随后的个体特征，是基因的结果。当然，基因与利他主义的关系是复杂的，可能有许多基因与环境因素相互作用，导致人们在不同情况下或多或少是无私的。尽管如此，几乎可以肯定的是基因导致人们变得无私。
- 最后，为什么呢？如果在自然选择下，动物为了生存和繁殖而彼此不断竞争，似乎显然利他主义会适得其反。在一个种群中，有些人帮助别人，甚至是为别人伤害自己，其他人纯粹是自私的，似乎自私者会受益，利他者会受到影响，并且利他主义的基因将被驱逐而灭绝。

> [3] 我希望你能原谅我在这里用“毫无疑问”代替实验的参考资料，我想在本章中介绍一些理由，而不会陷入太深。

这个明显的矛盾是“利他主义问题”：为什么利他主义的基因没有消失？

在生物学家中，有很多可能的解释，包括互惠利他主义，性选择，亲属选择和群体选择。而在非科学家中，还有更多的解释。我把它交给你去探索别的假说；现在我想专注于一种解释，可以说是最简单的一种解释：也许利他主义是适应性的。换句话说，利他主义的基因可能使人们更容易生存和繁殖。

事实证明，引发利他主义问题的囚徒困境，也可能有助于解决问题。

12.3 囚徒困境比赛

在 20 世纪 70 年代后期，密歇根大学的政治学家罗伯特阿克塞尔罗德（Robert Axelrod）组织了一场比赛来比较囚徒困境（PD）的策略。

他邀请参与者以计算机程序的形式提交策略，然后相互对抗并保持得分。具体来说，他们玩的是 PD 的迭代版本，其中智能体针对同一对手进行多轮比赛，因此他们的决定可以基于历史。

在 Axelrod 的比赛中，一个简单的策略出人意料地好，称为“针锋相对”，即 TFT，TFT 在第一轮迭代比赛中总是合作；之后，它会复制上一轮对手所做的任何事情。对手继续合作，TFT 保持合作，如果对手任何时候都背叛，下一轮 TFT 背叛，但如果对手变回合作，TFT 也会合作。

这些比赛的更多信息，以及 TFT 为何如此出色的解释，请参阅以下视频：https://www.youtube.com/watch?v=BOvAbjfJ0x0。

看看这些比赛中表现出色的策略，Alexrod 发现了他们倾向于分享的特点：

- 善良：表现好的策略在第一轮比赛中合作，并且通常会在随后的几轮中合作。
- 报复：始终合作的策略，并不如如果对手背叛就报复的策略好。
- 宽恕：但是过于斗气的策略往往会惩罚自己以及对手。
- 不嫉妒：一些最成功的策略很少超过对手；他们成功了，因为他们对各种各样的对手都做得足够好。

TFT 具有所有这些属性。

Axelrod 的比赛为利他主义问题提供了部分可能的答案：也许利他主义的基因是普遍存在的，因为它们是适应性的。 许多社会互动可以建模为囚徒困境的变种，就这种程度而言，如果将一个大脑设定为善良，平衡报复和宽恕，就会在各种各样的情况下表现良好。

但是 Axelrod 比赛中的策略是由人们设计的；他们并不进化。 我们需要考虑，善良、报复和宽恕的基因是否可以通过突变出现，成功侵入其他策略的种群，并抵制后续突变的侵入。

12.4 合作进化的模拟

合作进化是第一本书的标题，Axelrod 展示了来自囚徒困境比赛的结果，并讨论了利他主义问题的影响。 从那以后，他和其他研究人员已经探索了 PD 比赛的进化动态性，也就是说，PD 选手的总体中，策略的分布随时间如何变化。 在本章的其余部分中，我运行这些实验的一个版本并展示结果。

首先，我们需要一种将 PD 策略编码为基因型的方法。 在这个实验中，我考虑了一些策略，其中智能体每一轮的选择仅取决于前两轮中对手的选择。 我用字典来表示策略，它将对手的前两个选择映射为智能体的下一个选择。

以下是这些智能体的类定义：

```python
class Agent:

    keys = [(None, None),
            (None, 'C'),
            (None, 'D'),
            ('C', 'C'),
            ('C', 'D'),
            ('D', 'C'),
            ('D', 'D')]

    def __init__(self, values, fitness=np.nan):
        self.values = values
        self.responses = dict(zip(self.keys, values))
        self.fitness = fitness
```

`keys` 是每个智能体的词典中的键序列，其中元组 `('C', 'C')` 表示对手在前两轮合作；`(None, 'C')` 意味着只有一轮比赛并且对手合作；`(None, None)` 表示还没有回合。

在 `__init__` 方法中，`values` 是对应于键的一系列选项，`'C'` 或 `'D'`。 所以如果值的第一个元素是 `'C'`，那就意味着这个智能体将在第一轮合作。 如果值的最后一个元素是 `'D'`，那么如果对手在前两轮中背叛，该智能体将会背叛。

在这个实现中，总是背叛的智能体的基因型是 `'DDDDDDD'`; 总是合作的智能体的基因型是 `'CCCCCCC'`，而 TFT 的基因型是 `'CCDCDCD'`。

`Agent` 类提供 `copy`，它使其它智能体具有相同的基因型，但具有一定的变异概率：

```
prob_mutate = 0.05

def copy(self):
    if np.random.random() > self.prob_mutate:
        values = self.values
    else:
        values = self.mutate()
    return Agent(values, self.fitness)
```

突变的原理是，在基因型中选择一个随机值并从 `'C'` 翻转到 `'D'`，或者相反：

```
def mutate(self):
    values = list(self.values)
    index = np.random.choice(len(values))
    values[index] = 'C' if values[index] == 'D' else 'D'
    return values
```

既然我们有了智能体，我们还需要比赛。

12.5 `Tournament`

`Tournament` 类封装了 PD 比赛的细节：

```
payoffs = {('C', 'C'): (3, 3),
           ('C', 'D'): (0, 5),
           ('D', 'C'): (5, 0),
           ('D', 'D'): (1, 1)}

num_rounds = 6

def play(self, agent1, agent2):
    agent1.reset()
    agent2.reset()

    for i in range(self.num_rounds):
        resp1 = agent1.respond(agent2)
        resp2 = agent2.respond(agent1)

        pay1, pay2 = self.payoffs[resp1, resp2]

        agent1.append(resp1, pay1)
        agent2.append(resp2, pay2)

    return agent1.score, agent2.score
```

`payoffs` 是一个字典，将从智能体的选择映射为奖励。例如，如果两个智能体合作，他们每个得到 3 分。如果一个背叛而另一个合作，背叛者得到 5 分，而合作者得到 0 分。如果他们都背叛，每个都会得到 1 分。这些是 Axelrod 在他的比赛中使用的收益。

`play` 运行几轮 PD 游戏。它使用 `Agent` 类中的以下方法：

- `reset` ：在第一轮之前初始化智能体，重置他们的分数和他们的回应的历史记录。
- `respond` ：考虑到对手之前的回应，向每个智能体询问回应。
- `append` ：通过存储选项，并将连续轮次的分数相加，来更新每个智能体。

在给定的回合数之后， `play` 将返回每个智能体的总分数。我选择了 `num_rounds = 6` ，以便每个基因型的元素都以大致相同的频率访问。第一个元素仅在第一轮访问，或在六分之一的时间内访问。接下来的两个元素只能在第二轮中访问，或者每个十二分之一。最后四个元素在六分之一时间内访问，平均每次访问六次，或者平均每个六分之一。

`Tournament` 提供了第二种方法，即 `melee` ，确定哪些智能体互相竞争：

```
def melee(self, agents, randomize=True):
    if randomize:
        agents = np.random.permutation(agents)

    n = len(agents)
    i_row = np.arange(n)
    j_row = (i_row + 1) % n

    totals = np.zeros(n)

    for i, j in zip(i_row, j_row):
        agent1, agent2 = agents[i], agents[j]
        score1, score2 = self.play(agent1, agent2)
        totals[i] += score1
        totals[j] += score2

    for i in i_row:
        agents[i].fitness = totals[i] / self.num_rounds / 2
```

`melee` 接受一个智能体列表和一个布尔值 `randomize` ，它决定了每个智能体每次是否与同一邻居竞争，或者匹配是否随机化。

`i_row` 和 `j_row` 包含匹配的索引。 `totals` 包含每个智能体的总分数。

在循环内部，我们选择两个智能体，调用 `play` 和更新 `totals` 。 最后，我们计算每个智能体获得的，每轮和每个对手的平均点数，并将结果存储在每个智能体的 `fitness` 属性中。

12.6 `Simulation`

本章的 `Simulation` 类基于第？章的中的那个；唯一的区别是 `__init__` 和 `step` 。

这是 `__init__` 方法：

```
class PDSimulation(Simulation):

    def __init__(self, tournament, agents):
        self.tournament = tournament
        self.agents = np.asarray(agents)
        self.instruments = []
```

`Simulation` 对象包含一个 `Tournament` 对象，一系列的智能体和一系列的 `Instrument` 对象（就像第？章中一样）。

以下是 `step` 方法：

```
def step(self):
    self.tournament.melee(self.agents)
    Simulation.step(self)
```

此版本的 `step` 使用 `Tournament.melee` ，它为每个智能体设置 `fitness` 属性；然后它调用父类的 `step` 方法，父类来自第？章：

```
# class Simulation

    def step(self):
        n = len(self.agents)
        fits = self.get_fitnesses()

        # see who dies
        index_dead = self.choose_dead(fits)
        num_dead = len(index_dead)

        # replace the dead with copies of the living
        replacements = self.choose_replacements(num_dead, fits)
        self.agents[index_dead] = replacements

        # update any instruments
        self.update_instruments()
```

`Simulation.step` 将智能体的适应性收集到一个数组中; 然后它会调用 `choose_dead` 来决定哪些智能体死掉，并用 `choose_replacements` 来决定哪些智能体繁殖。

我的模拟包含生存差异，就像第？章那样，但不包括繁殖差异。 你可以在本章的笔记本上看到细节。 作为练习之一，你将有机会探索繁殖差异的效果。

12.7 结果

假设我们从三个智能体开始：一个总是合作，一个总是背叛，另一个执行 TFT 策略。 如果我们在这个种群中运行 `Tournament.melee`，合作者每轮获得 1.5 分，TFT 智能体获得 1.9 分，而背叛者获得 3.33 分。 这个结果表明，“总是背叛”应该很快成为主导策略。

但是“总是缺陷”包含着自我破坏的种子，如果更好的策略被驱使而灭绝，那么背叛者就没有人可以利用，他们的适应性下降，并且容易受到合作者的入侵。

根据这一分析，预测系统的行为不容易：它会找到一个稳定的平衡点，还是在基因型景观的各个位置之间振荡？ 让我们运行模拟来发现它！

我以 100 个始终背叛的相同智能体开始，并运行 5000 个步骤的模拟：

```
tour = Tournament()
agents = make_identical_agents(100, list('DDDDDDD'))
sim = PDSimulation(tour, agents)
```

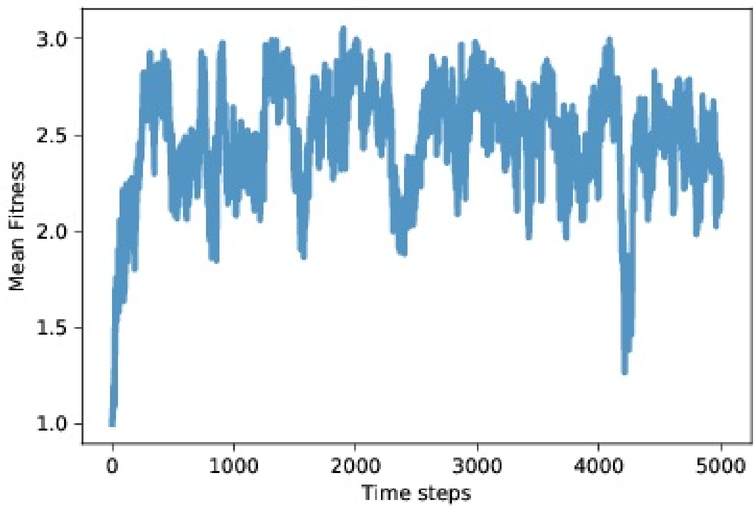


图 12.1：平均适应性（囚徒困境的每个回合的所得点数）

图？展示了随时间变化的平均适应性（使用第？章的 `MeanFitness` 仪器）。最初平均适应性是 1，因为当背叛者面对对方时，他们每轮只能得到 1 分。

经过大约 500 个时间步，平均适应性增加到近 3，这是合作者面对彼此时得到的。但是，正如我们所怀疑的那样，这种情况不稳定。在接下来的 500 个步骤中，平均适应性下降到 2 以下，回到 3，并继续振荡。

模拟的其余部分变化很大，但除了一次大的下降之外，平均适应性通常在 2 到 3 之间，长期平均值接近 2.5。

而且这还不错！它不是一个合作的乌托邦，每轮平均得 3 分，但距离始终背叛的乌托邦还很远。这比我们所期待的，自利智能体的自然选择要好得多。

为了深入了解这种适应性水平，我们来看看更多的仪器。`Niceness` 在每个时间步骤之后测量智能体的基因型的合作比例：

```
class Niceness(Instrument):

    def update(self, sim):
        responses = np.array([agent.values
                              for agent in sim.agents])
        metric = np.mean(responses == 'C')
        self.metrics.append(metric)
```

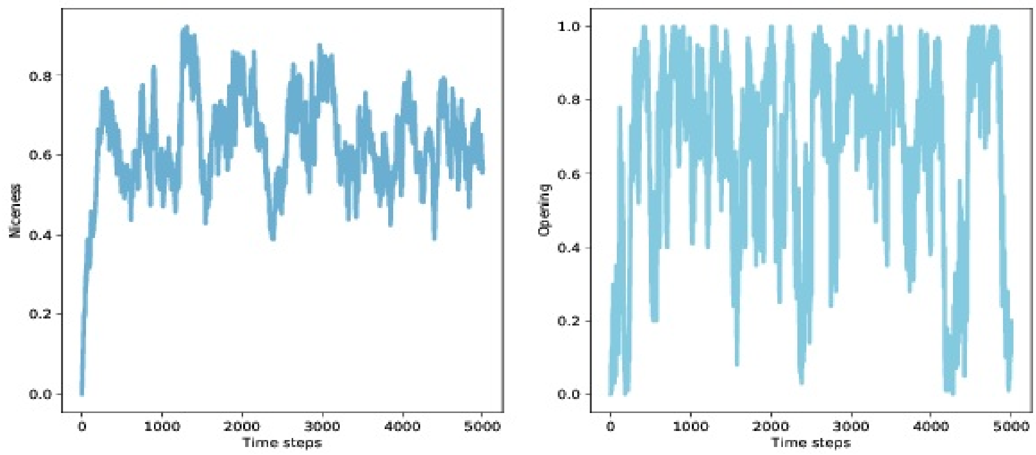


图 12.2：种群中所有基因组的平均友善度（左）和第一轮合作的种群比例（右）

图？（左图）展示结果：平均友善度从 0 迅速上升到 0.75，然后在 0.4 到 0.85 之间波动，长期平均值接近 0.65。 同样，这相当好！

具体看开始的移动，我们可以追踪第一轮合作的智能体的比例。 这是这个仪器：

```
class Retaliating(Instrument):

    def update(self, sim):
        after_d = np.array([agent.values[2::2]
                            for agent in sim.agents])
        after_c = np.array([agent.values[1::2]
                            for agent in sim.agents])
        metric = np.mean(after_d=='D') - np.mean(after_c=='D')
        self.metrics.append(metric)
```

报复行为将所有基因组中的元素数量，其中对手背叛后智能体也背叛（元素 2, 4 和 6），与其中的元素数量，其中对手合作后智能体背叛相比较。正如你现在的预期，结果差异很大（你可以在笔记本中看到图形）。平均而言，这些分数之间的差异小于 0.1，所以如果智能体在对手合作后，30% 的时间中背叛，他们可能会在背叛后的 40% 时间中背叛。

这个结果为这个断言提供了较弱的支持，即成功的策略会报复。也许所有智能体甚至很多智能体都没有必要进行报复；如果整个种群中至少存在一定的报复倾向，那么这可能足以阻止高度报复策略的普及。

为了衡量宽恕，我再次定义了一个工具，来查看在前两轮之后，智能体是否更有可能在 D-C 之后进行合作，与 C-D 相比。在我的模拟中，没有证据表明这种特殊的宽恕。另一方面，这些模拟中的策略在某种意义上是必然的宽容，因为它们只考虑前两轮的历史。

12.8 总结

Axelrod 的比赛提出了解决利他主义问题的一个可能的解决办法：或许善良，但不是太善良，是适应性的。但是原始比如中的策略是由人们，而不是进化论设计的，并且策略的分布在比赛过程中没有改变。

所以这就提出了一个问题：像 TFT 这样的策略可能会在固定的人为设计策略中表现良好，但它们是否会进化？换句话说，他们是否可以通过变异出现在种群中，与祖先竞争成功，并抵抗他们的后代的入侵？

本章中的模拟表明：

- 背叛者种群容易受到更善良的策略的入侵。
- 过于善良的种群容易受到背叛者的入侵。
- 所以，善良的平均程度有所波动，但善良的平均数量普遍较高，而平均适应程度一般更接近合作乌托邦而不是偏差异议程度。
- 在 Alexrod 的比赛中，TFT 是一项成功的战略，但对于不断发展的种群来说，这似乎不是一个最佳策略。事实上，可能没有稳定的最佳策略。
- 某种程度的报复可能是适应性的，但对所有智能体来说，可能没有必要进行报复。 如果在整个种群中有足够的报复行为，这可能足以防止背叛者入侵 [4]。

> [4] 这就引入了博弈论中一个全新的话题 - 搭便车问题（见 https://en.wikipedia.org/wiki/Free-rider_problem）。

很明显，这些模拟中的智能体很简单，而囚徒困境是一种有限范围的社交互动的高度抽象模型。 尽管如此，本章的结果对人性提供了一些见解。 也许我们对合作，报复和宽恕的倾向是天生的，至少部分是。 这些特征是我们的大脑的工作机制的结果，至少部分是由我们的基因控制的。 也许我们的基因这样来构建我们的大脑，因为在人类进化史上，自私的大脑的基因不太可能传播。

所以也许这就是为什么自私基因会建立无私的大脑。

12.9 练习

本章的代码位于本书仓库的 Jupyter 笔记本 `chap12.ipynb` 中。打开笔记本，阅读代码并运行单元个。你可以使用这个笔记本来练习本章的练习。我的解决方案在 `chap12soln.ipynb` 中。

练习 1

本章中的模拟取决于我任意选择的条件和参数。作为练习，我鼓励你去探索其他条件，看看他们对结果有什么影响。这里有一些建议：

1. 改变初始条件：不要从所有背叛者开始，看看如果从所有合作者，所有 TFT 或随机智能体开始会发生什么。
2. 在 `Tournament.melee` 中，我在每个时间步骤开始时洗牌，所以每个玩家对抗两个随机选择的玩家。如果你不洗牌会怎么样？在这种情况下，每个智能体都会反复与相同的邻居进行比赛。这可能会让少数人的战略，更容易通过利用局部性入侵大多数。
3. 由于每个智能体只与另外两个智能体进行比赛，因此每轮比赛的结果都是非常不同的：在任何一轮比赛中，胜过大部分智能体的智能体可能会运气不好，或者相反。如果增加每个智能体在每轮中的对手数量会发生什么？或者如果在每一步结束时，智能体的适应性是上一轮结束时其当前得分和适应性的平均值，会怎么样？
4. 我为 `prob_survival` 函数选择的值从 0.7 到 0.9 不等，所以适应性最差的智能体 `p = 0.7`，生存了 3.33 个时间步骤，适应性最强的智能体生存了 10 个。如果你使 `prob_survival` 更加或更加不“激进”，会发生什么情况。
5. 我选择了 `num_rounds = 6`，以便基因组的每个元素对比赛的结果具有大致相同的影响。但这比 Alexrod 在他的比赛中使用的值要小得多。 如果增加 `num_rounds` 会发生什么？ 注意：如果你研究这个参数的效果，你可能想修改 `Niceness` 来衡量基因组中最后4个元素的友善度，随着 `num_rounds` 的增加，它会受到更多的选择性压力。
6. 我的实现拥有生存差异和随机繁殖。 如果添加繁殖差异会发生什么？

练习 2

在我的模拟中，种群从未收敛到一个状态，其中多数人共享相同的，据推测是最佳的基因型。对于这个结果有两种可能的解释：一是没有最佳策略，因为无论何时种群被大多数基因型控制，这种状况为少数人入侵提供了机会；另一种可能性是，突变率高到足以维持多种基因型，即使多数是非最佳的。为了辨别这些解释，请尝试降低突变率来查看发生了什么。或者，从随机种群开始，并且不带突变来运行，直到只有一个基因型存活。或者带突变来运行，直到系统达到稳定状态；然后关闭突变并运行，直到只有一个幸存的基因型。这些情况下基因型的特征是什么？

练习 3

我的实验中的智能体是“反应型”的，因为他们在每轮中的选择只取决于对手在前几轮中的做法。考虑探索一些策略，它们也考虑到智能体过去的选择。这样的策略将能够区分报复性对手，和没有挑衅而背叛的对手。

附录 A、算法分析

> 原文：Appendix A Analysis of algorithms
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译
>
> 部分参考了《Think Python 2e 中译本 第二十一章：算法分析》

算法分析 (**Analysis of algorithms**) 是计算机科学的一个分支， 着重研究算法的性能， 特别是它们的运行时间和资源开销。见 http://en.wikipedia.org/wiki/Analysis_ofalgorithms 。

算法分析的实际目的是预测不同算法的性能，用于指导设计决策。

2008年美国总统大选期间，当候选人奥巴马(Barack Obama)访问Google时， 他被要求进行即时分析。首席执行官 Eric Schmidt 开玩笑地问他“对一百万个32位整数排序的最有效的方法”。 显然有人暗中通知了奥巴马，因为他很快回答，“我认为不应该采用冒泡排序法”。 详见 http://www.youtube.com/watch?v=k4RRi_ntQc8 。

是真的：冒泡排序概念上很简单，但是对于大数据集来说速度非常慢。Schmidt所提问题的答案可能是 “基数排序 (http://en.wikipedia.org/wiki/Radix_sort)” [1]。

> [1] 但是，如果你面试中被问到这个问题，我认为更好的答案是，“对上百万个整数进行最快排序的方法就是用你所使用的语言的内建排序函数。它的性能对于大多数应用而言已优化的足够好。但如果最终我的应用运行太慢，我会用性能分析器找出大量的运算时间被用在了哪儿。如果采用一个更快的算法会对性能产生显著的提升，我会试着找一个基数排序的优质实现。”

算法分析的目的是在不同算法间进行有意义的比较， 但是有一些问题：

- 算法的相对性能依赖于硬件的特性，因此一个算法可能在机器A上比较快， 另一个算法则在机器B上比较快。 对此问题一般的解决办法是指定一个 机器模型 (machine model) 并且分析一个算法在一个给定模型下所需的步骤或运算的数目。
- 相对性能可能依赖于数据集的细节。 例如， 如果数据已经部分排好序， 一些排序算法可能更快； 此时其它算法运行的比较慢。 避免该问题的一般方法是分析 最坏情况。 有时分析平均情况性能也可， 但那通常更难，而且可能不容易弄清该对哪些数据集合进行平均。
- 相对性能也依赖于问题的规模。一个对于小列表很快的排序算法可能对于长列表很慢。此问题通常的解决方法是将运行时间（或者运算的次数）表示成问题规模的函数， 并且根据各自随着问题规模的增长而增加的速度，将函数分成不同的类别。

此类比较的好处是有助于对算法进行简单的分类。 例如，如果我知道算法A的运行时间与输入的规模 `n` 成正比， 算法 B 与 `n^2` 成正比，那么我可以认为 A 比 B 快，至少对于很大的 `n` 值来说。

这类分析也有一些问题，我们后面会提到。

A.1 增长级别

假设你已经分析了两个算法，并能用输入计算量的规模表示它们的运行时间： 若算法 A 用 `100n+1` 步解决一个规模为 `n` 的问题；而算法 B 用 `n^2 + n + 1` 步。

下表列出了这些算法对于不同问题规模的运行时间：

输入大小	算法 A 的运行时间	算法 B 的运行时
10	1,001	111
100	10,001	10,101
1,000	100,001	1,001,001
10,000	1,000,001	`> 10^10`

当 `n=10` 时，算法 A 看上去很糟糕，它用了 10 倍于算法 B 所需的时间。 但当 `n=100` 时 ，它们性能几乎相同， 而 `n` 取更大值时，算法 A 要好得多。

根本原因是对于较大的 `n` 值，任何包含 `n^2` 项的函数都比首项为 `n` 的函数增长要快。 首项 (leading term) 是指具有最高指数的项。

对于算法A，首项有一个较大的系数 100，这是为什么对于小 `n` ，B比A好。但是不考虑该系数，总有一些 `n` 值使得 `a n^2 > b n`，`a` 和 `b` 可取任意值。

同样推论也适用于非首项。 即使算法 A 的运行时间为 `n+1000000` ，对于足够大的 `n` ，它仍然比算法 B 好。

一般来讲，我们认为具备较小首项的算法对于规模大的问题是一个好算法，但是对于规模小的问题，可能存在有一个 交叉点 (crossover point)，在此规模以下，另一个算法更好。 交叉点的位置取决于算法的细节、输入以及硬件，因此在进行算法分析时它通常被忽略。 但是这不意味着你可以忘记它。

如果两个算法有相同的首项，很难说哪个更好；答案还是取决于细节。 所以对于算法分析来说，具有相同首项的函数被认为是相当的，即使它们具有不同的系数。

增长级别(order of growth)是一个函数集合，集合中函数的增长行为被认为是相当的。 例如 `2n` 、 `100n` 和 `n+1` 属于相同的增长级别，可用 大*O*符号(Big-Oh notation) 写成 `O(n)` ， 而且常被称作 线性级 (linear)，因为集合中的每个函数随着 `n` 线性增长。

首项为 `n^2` 的函数属于 `O(n^2)` ；它们被称为 二次方级 (quadratic)。

下表列出了算法分析中最通常的一些增长级别，按照运行效率从高到低排列。

增长级别	名称
`O(1)`	常数
`O(logn)`	对数
`O(n)`	线性
`O(n logn)`	线性对数
`O(n^2)`	二次
`O(n^3)`	三次
`O(c^n)`	指数

对于对数级，对数的基数并不影响增长级别。 改变基数等价于乘以一个常数，其不改变增长级别。相应的，所有的指数级数都属于相同的增长级别，而无需考虑指数的基数大小。指数函数增长级别增长的非常快，因此指数级算法只用于小规模问题。

练习 1

访问 http://en.wikipedia.org/wiki/Big_O_notation ，阅读维基百科关于大O符号的介绍，并回答以下问题：

1. `n^3 + n^2` 的增长级别是多少？`1000000 n^3 + n^2` 和 `n^3 + 1000000 n^2` 的增长级别又是多少？
2. `(n^2 + n) * (n + 1)` 的增长级别是多少？在开始计算之前，记住你只需要考虑首项即可。
3. 如果 `f` 的增长级别为 `O(g)` ，那么对于未指定的函数 `g` ，我们可以如何描述 `af+b` ？
4. 如果 `f1` 和 `f2` 的增长级别为 `O(g)`，那么 `f1 + f2` 的增长级别又是多少？
5. 如果 `f1` 的增长级别为 `O(g)` ，`f2` 的增长级别为 `O(h)`，那么 `f1 + f2` 的增长级别是多少？
6. 如果 `f1` 的增长级别为 `O(g)` ，`f2` 的增长级别为 `O(h)`，那么 `f1 * f2` 的增长级别是多少？

关注性能的程序员经常发现这种分析很难忍受。他们的观点有一定道理：有时系数和非首项会产生巨大的影响。 有时，硬件的细节、编程语言以及输入的特性会造成很大的影响。对于小问题，渐近的行为没有什么影响。

但是，如果你牢记这些注意事项，算法分析就是一个有用的工具。 至少对于大问题，“更好的” 算法通常更好，并且有时要好的多。 相同增长级别的两个算法之间的不同通常是一个常数因子，但是一个好算法和一个坏算法之间的不同是无限的！

A.2 Python基本运算操作分析

在 Python 中，大部分算术运算的开销是常数级的；乘法会比加减法用更长的时间，除法更长， 但是这些运算时间不依赖被运算数的数量级。非常大的整数却是个例外；在这种情况下，运行时间随着位数的增加而增加。

索引操作 — 在序列或字典中读写元素 — 的增长级别也是常数级的，和数据结构的大小无关。

一个遍历序列或字典的 for 循环通常是线性的，只要循环体内的运算是常数时间。 例如，累加一个列表的元素是线性的：

```
total = 0
for x in t:
    total += x
```

内建函数 `sum` 也是线性的，因为它做的是相同的事情，但是它要更快一些，因为它是一个更有效的实现；从算法分析角度讲，它具有更小的首项系数。

根据经验，如果循环体内的增长级别是 `O(n^a)` ，则整个循环的增长级别是 `O(n^(a+1))` 。如果这个循环在执行一定数目循环后退出则是例外。 无论 `n` 取值多少，如果循环仅执行 `k` 次， 整个循环的增长级别是 `O(n^a)` ，即便 `k` 值比较大。

乘上 `k` 并不会改变增长级别，除法也是。 因此，如果循环体的增长级别是 `O(n^a)` ，而且循环执行 `n/k` 次，那么整个循环的增长级别就是 `O(n^(a+1))` , 即使 `k` 值很大。

大部分字符串和元组运算是线性的，除了索引和 `len` ，它们是常数时间。 内建函数 `min` 和 `max` 是线性的。切片运算与输出的长度成正比，但是和输入的大小无关。

字符串拼接是线性的；它的运算时间取决于运算对象的总长度。

所有字符串方法都是线性的，但是如果字符串的长度受限于一个常数 — 例如，在单个字符上的运算 — 它们被认为是常数时间。字符串方法 `join` 也是线性的；它的运算时间取决于字符串的总长度。

大部分的列表方法是线性的，但是有一些例外：

- 平均来讲，在列表结尾增加一个元素是常数时间。 当它超出了所占用空间时，它偶尔被拷贝到一个更大的地方，但是对于 `n` 个运算的整体时间仍为 `O(n)` ， 所以我每个运算的平均时间是 `O(1)` 。
- 从一个列表结尾删除一个元素是常数时间。
- 排序是 `O(n logn)` 。

大部分的字典运算和方法是常数时间，但有些例外：

- `update` 的运行时间与作为形参被传递的字典（不是被更新的字典）的大小成正比。
- `keys` 、 `values` 和 `items` 是常数时间，因为它们返回迭代器。 但是如果你对迭代器进行循环，循环将是线性的。

字典的性能是计算机科学的一个小奇迹之一。在哈希表一节中，我们将介绍它们是如何工作的。

练习 2

访问 http://en.wikipedia.org/wiki/Sorting_algorithm ，阅读维基百科上对排序算法的介绍，并回答下面的问题：

1. 什么是“比较排序”？比较排序在最差情况下的最好增长级别是多少？别的排序算法在最差情况下的最优增长级别又是多少？
2. 冒泡排序法的增长级别是多少？为什么奥巴马认为是“不应采用的方法”
3. 基数排序(radix sort)的增长级别是多少？我们使用它之前需要具备的前提条件有哪些？
4. 排序算法的稳定性是指什么？为什么它在实际操作中很重要？
5. 最差的排序算法是哪一个（有名称的）？
6. C 语言使用哪种排序算法？Python使用哪种排序算法？这些算法稳定吗？你可能需要谷歌一下，才能找到这些答案。
7. 大多数非比较算法是线性的，因此为什 Python 使用一个 增长级别为 `O(n logn)` 的比较排序？

A.3 搜索算法分析

搜索 (search)算法，接受一个集合以及一个目标项，并判断该目标项是否在集合中，通常返回目标的索引值。

最简单的搜索算法是“线性搜索”，其按顺序遍历集合中的项，如果找到目标则停止。 最坏的情况下， 它不得不遍历全部集合，所以运行时间是线性的。

序列的 in 操作符使用线性搜索；字符串方法 `find` 和 `count` 也使用线性搜索。

如果元素在序列中是排序好的，你可以用 二分搜素 (bisection search) ，它的增长级别是 `O(logn)` 。 二分搜索和你在字典中查找一个单词的算法类似（这里是指真正的字典，不是数据结构）。 你不会从头开始并按顺序检查每个项，而是从中间的项开始并检查你要查找的单词在前面还是后面。 如果它出现在前面，那么你搜索序列的前半部分。否则你搜索后一半。如论如何，你将剩余的项数分为一半。

练习 3

编写一个叫做 `bisection` 的函数，它接受有序列表和目标值，并返回列表中值的索引（如果存在的话）；如果不存在则返回 `None` 。

或者你可以阅读对分模块的文档并使用它！

如果序列有 1,000,000 项，它将花 20 步找到该单词或判断出其不在序列中。因此它比线性搜索快大概 50,000 倍。

二分搜索比线性搜索快很多，但是它要求已排序的序列，因此使用时需要做额外的工作。

另一个检索速度更快的数据结构被称为 哈希表 (hashtable) — 它可以在常数时间内检索出结果 — 并且不依赖于序列是否已排序。 Python 中的字典就通过哈希表技术实现的，因此大多数的字典操作，包括 in 操作符，只花费常数时间就可完成。

A.4 哈希表

为了解释哈希表是如何工作以及为什么它的性能如此优秀， 我们从实现一个简单的映射(map)开始并逐步改进它，直到其成为一个哈希表。

我们使用 Python 来演示这些实现，但在现实生活中，你用不着用 Python 写这样的代码；你只需用内建的字典对象就可以了！因此在接下来的内容中，你就当字典对象并不存在，你希望自己实现一个将键映射到值的数据结构。你必须实现的操作包括：

`add(k, v)` ：

> 增加一个新的项，其从键 k 映射到值 v 。 如果使用 Python 的字典d，该运算被写作 `d[k] = v` 。

`get(k)` ：

> 查找并返回相应键的值。 如果使用 Python 的字典d，该运算被写作 `d[k]` 或 `d.get(k)` 。

现在，假设每个键只出现一次。该接口最简单的实现是使用一个元组列表，其中每个元组是一个键-值对。

```
class LinearMap:

    def __init__(self):
        self.items = []

    def add(self, k, v):
        self.items.append((k, v))

    def get(self, k):
        for key, val in self.items:
            if key == k:
                return val
        raise KeyError
```

`add` 向项列表追加一个键—值元组，其增长级别为常数时间。

`get` 使用 `for` 循环搜索该列表：如果它找到目标键，则返回相应的值；否则触发一个 `KeyError` 。因此 `get` 是线性的。

另一个方案是保持列表按键排序。那么，`get` 可以使用二分搜索，其增长级别为 `O(logn)`。但是在列表中间插入一个新的项是线性的，因此这可能不是最好的选择。有其它的数据结构能在对数级时间内实现 `add` 和 `get`，但是这仍然不如常数时间好，那么我们继续。

另一种改良 `LinearMap` 的方法是将键-值对列表分成小列表。下面是一个被称作 `BetterMap` 的实现，它是 100 个 `LinearMap` 组成的列表。正如一会儿我们将看到的，`get` 的增长级别仍然是线性的，但是 `BetterMap` 是迈向哈希表的一步。

```
class BetterMap:

    def __init__(self, n=100):
        self.maps = []
        for i in range(n):
            self.maps.append(LinearMap())

    def find_map(self, k):
        index = hash(k) % len(self.maps)
        return self.maps[index]

    def add(self, k, v):
        m = self.find_map(k)
        m.add(k, v)

    def get(self, k):
        m = self.find_map(k)
        return m.get(k)
```

`__init__` 会生成一个由 n 个 `LinearMap` 组成的列表。

`add` 和 `get` 使用 `find_map` 查找往哪一个列表中添加新项，或者对哪个列表进行检索。

`find_map` 使用了内建函数 `hash`，其接受几乎任何 Python 对象并返回一个整数。这一实现的一个限制是它仅适用于可哈希的键。像列表和字典等可变类型是不能哈希的。

被认为是相等的可哈希对象返回相同的哈希值，但是反之不是必然成立：两个具备不同值的对象能够返回相同的哈希值。

`find_map` 使用求余运算符将哈希值包在 0 到 `len(self.maps)` 之间，因此结果是该列表的合法索引值。当然，这意味着许多不同的哈希值将被包成相同的索引值。但是如果哈希函数散布相当均匀（这是哈希函数被设计的初衷），那么我们预计每个 `LinearMap` 会有 `n/100` 项。

由于 `LinearMap.get` 的运行时间与项数成正比，那么我们预计 `BetterMap` 比 `LinearMap` 快100倍。增长级别仍然是线性的，但是首项系数变小了。这样很好，但是仍然不如哈希表好。

下面是使哈希表变快的关键：如果你能保证 `LinearMap` 的最大长度是有上限的，则 `LinearMap.get` 的增长级别是常数时间。你只需要跟踪项数并且当每个 `LinearMap` 的项数超过阈值时，通过增加更多的 `LinearMap` 调整哈希表的大小。

以下是哈希表的一个实现：

```
class HashMap:

    def __init__(self):
        self.maps = BetterMap(2)
        self.num = 0

    def get(self, k):
        return self.maps.get(k)

    def add(self, k, v):
        if self.num == len(self.maps.maps):
            self.resize()

        self.maps.add(k, v)
        self.num += 1

    def resize(self):
        new_maps = BetterMap(self.num * 2)

        for m in self.maps.maps:
            for k, v in m.items:
                new_maps.add(k, v)

        self.maps = new_maps
```

每个 `HashMap` 包含一个 `BetterMap` 。 `__init__` 开始仅有两个 `LinearMap` ，并且初始化 `num` ，用于跟踪项的数量。

`get` 仅仅用来调度 `BetterMap` 。真正的操作发生于 `add` 内，其检查项的数量以及 `BetterMap` 的大小： 如果它们相同，每个 `LinearMap` 的平均项数为 1，因此它调用 `resize` 。

`resize` 生成一个新的 `BetterMap` ，是之前那个的两倍大，然后将像从旧表“重新哈希”至到新的表。

重新哈希是必要的，因为改变 `LinearMap` 的数目也改变了 `find_map` 中求余运算的分母。 这意味着一些被包进相同的 `LinearMap` 的对象将被分离（这正是我们希望的，对吧？）。

重新哈希是线性的，因此 `resize` 是线性的，这可能看起来很糟糕，因为我保证 `add` 会是常数时间。 但是记住，我们不必每次都调整，因此 `add` 通常是常数时间，只是偶尔是线性的。 运行 `add` `n` 次的整体操作量与 `n` 成正比，因此 `add` 的平均运行时间是常数时间！

为了弄清这是如何工作的，考虑以一个空的 `HashTable` 开始并增加一系列项。 我们以两个 `LinearMap` 开始，因此前两个 `add` 操作很快（不需要调整大小）。 我们假设它们每个操作花费一个工作单元。下一个 `add` 需要进行一次大小调整， 因此我们必须重新哈希前两项（我们将其算成两个额外的工作单元），然后增加第3项（又一个工作单元）。 增加下一项的花费一个单元，所以目前为止添加四个项共需要 6 个单元。

下一个 `add` 花费 5 个单元，但是之后的3个操作每个只花费 1 个单元，所以前八个 `add` 总共需要 14 个单元。

下一个 `add` 花费 9 个单元，但是之后在下一次调整大小之前，可以再增加七个， 所以前 16 个 `add` 总共需要 30 个单元。

进行 32 次 `add` 之后，总共花费了 62 个单元，我希望你开始看到规律。 `n` 次 `add` 后，其中 `n` 是 2 的倍数，总花费是 `2n-2` 个单元， 所以平均每个 `add` 操作只花费了少于 2 个单元。当 `n` 是 2 的倍数时，那是最好的情况。 对于其它的 `n` 值，平均花费稍高一点，但是那并不重要。重要的是其增长级别为 `O(1)` 。

下图形象地说明了其工作原理。每个区块代表一个工作单元。 每列显示每个 `add` 所需的单元，按从左到右的顺序排列：前两个 `add` 花费 1 个单元，第三个花费 3 个单元，等等。

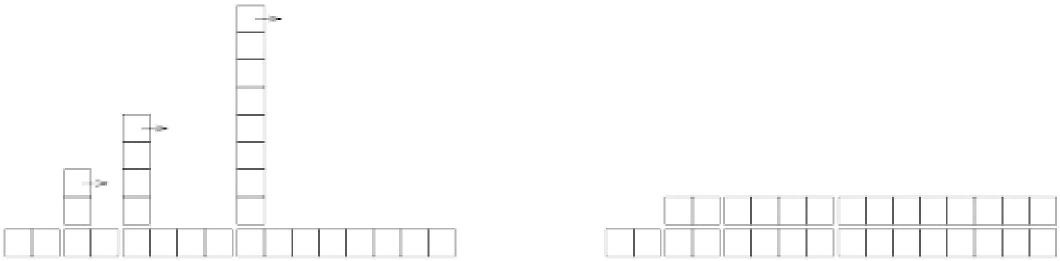


图 A.1：哈希表中 `add` 操作的成本

重新哈希的额外工作，表现为一系列不断增高的高塔，各自之间的距离越来越大。 现在，如果你打翻这些塔，将大小调整的代价均摊到所有的 `add` 上，你会从图上看到 `n` 次 `add` 的整个花费是 `2n - 2` 。

该算法一个重要的特征是，当我们调整 `HashTable` 的大小时，它呈几何级增长；也就是说，我们用常数乘以表的大小。 如果你按算术级增加大小 —— 每次增加固定的数目 —— 每个 `add` 的平均时间是线性的。

你可以从 http://thinkpython2.com/code/Map.py 下载到 `HashMap` 的实现代码，你不必使用它；如果你想要一个映射数据结构，只要使用 Python 中的字典即可。

练习 4

我的 `HashMap` 实现直接访问 `BetterMap` 的属性，这表现了糟糕的面向对象设计。

- 特殊方法 `__len__` 由内置函数 `len` 调用。 为 `BetterMap` 编写一个 `__len__` 方法并在 `add` 中使用它。
- 使用生成器来编写 `BetterMap.iteritems` ，并在 `resize` 中使用它。

练习 5

散列表的一个缺点是元素必须是可散列的，这通常意味着它们必须是不可变的。 这就是为什么在 Python 中，可以将元组而不是列表用作字典中的键。 另一种方法是使用基于树的映射。

编写一个名为 `TreeMap` 的映射接口的实现，它使用红黑树，以对数时间执行 `add` 和 `log` 。

A.5 列表的求和

假设你有一堆列表，并且你想把它们合并成一个列表。 有三种方法可以在 Python 中执行此操作：

你可以使用 `+=` 运算符：

```
total = []
for x in t:
    total += x
```

或者 `extend` 方法：

```
total = []
for x in t:
    total.extend(x)
```

或者内建的 `sum` 函数：

```
total = sum(t, [])
```

`sum` 的第二个参数是总数的初始值。

在不知道如何实现 `+=` 和 `extend` 和 `sum` 的情况下，很难分析它们的性能。 例如，如果 `total += x` 每次创建一个新列表，则循环是二次的；但如果它修改了总数，它是线性的。

为了找到答案，我们可以阅读源代码，但作为练习，让我们看看我们是否可以通过测量运行时间来弄清楚它。

测量程序运行时间的简单方法，是使用 `os` 模块中的 `time` 函数，该函数返回浮点数的元组，表示进程已经过的时间（详细信息请参阅文档）。 我使用了函数 `etime` ，它返回"用户时间"和"系统时间"的总和，这通常是我们关心的性能度量：

```
import os

def etime():
    """See how much user and system time this process has used
    so far and return the sum."""

    user, sys, chuser, chsys, real = os.times()
    return user+sys
```

为了衡量一个函数的运行时间，你可以调用 `etime` 两次并计算差异：

```
start = etime()

# put the code you want to measure here

end = etime()
elapsed = end - start
```

或者，如果你使用 IPython，则可以使用 `timeit` 命令。 请参阅 `ipython.scipy.org` 。

如果算法是二次的，我们期望运行时间 `t` 与输入大小 `n` 的函数，是这样的：

```
t = a * n^2 + b * n + c
```

其中 `a` ，`b` 和 `c` 是未知系数。 如果你对两边取对数，你会得到：

```
logt ~ loga + 2logn
```

对于 `n` 的较大值，非主要项是微不足道的，并且这个近似值非常好。 所以如果我们在双对数刻度上绘制 `t` 对 `n` ，我们期待斜率为 2 的直线。

类似地，如果算法是线性的，我们期望斜率为 1 的直线。

我写了三个连接列表的函数： `sum_plus` 使用 `+=` ； `sum_extend` 使用 `list.extend` ； `sum_sum` 使用 `sum` 。 我在 `n` 的范围内对它们计时，并将结果绘制在双对数刻度上。 下图展示了结果。

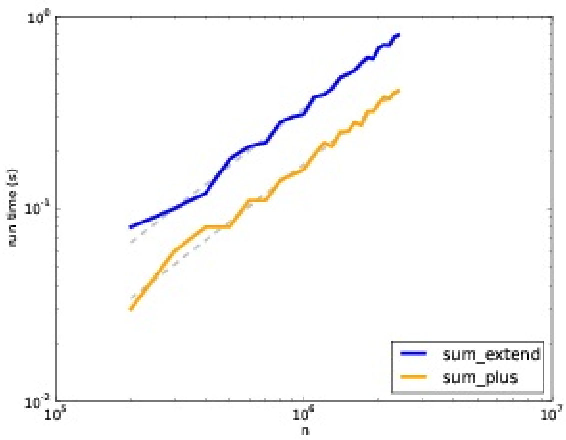


图 a.2：运行时间和 `n` ，虚线斜率为 1

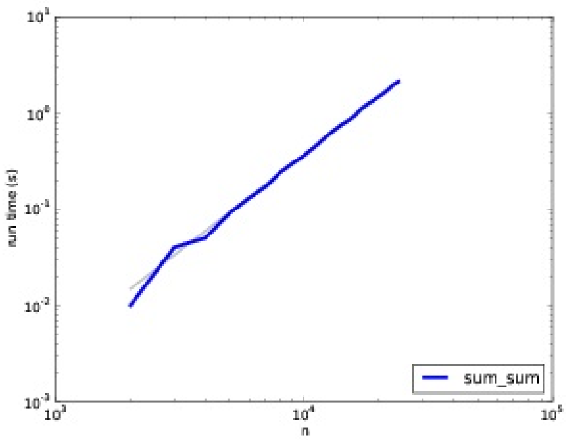


图 a.3：运行时间和 `n` ，虚线斜率为 2

在图 a.2 中，我用斜率为 1 的直线拟合了曲线。 这条线很好地拟合了数据，所以我们得出结论，这些实现是线性的。 `+=` 实现的速度比较快，因为每次循环中，查找 `extend` 方法需要一些时间。

在图 a.3 中，斜率 2 的线拟合了数据，所以 `sum` 实现是二次的。

A.6 `pyplot`

为了制作本节中的图片，我使用了 `pyplot` ，它是 `matplotlib` 的一部分。 如果你的 Python 安装没有带着 `matplotlib` ，你可能需要安装它，或者你可以使用另一个库进行绘图。

下面是一个简单的例子：

```
import matplotlib.pyplot as pyplot

pyplot.plot(xs, ys)
scale = 'log'
pyplot.xscale(scale)
pyplot.yscale(scale)
pyplot.title('')
pyplot.xlabel('n')
pyplot.ylabel('run time (s)')
pyplot.show()
```

导入语句使 `matplotlib.pyplot` 可以使用较短的名称 `pyplot` 访问。

`plot` 接受 `x` 值列表和一个 `y` 值列表并绘制它们。 列表的长度必须相同。 `xscale` 和 `yscale` 设置线性或对数轴。

`title` ， `xlabel` 和 `ylabel` 是不言自明的。 最后， `show` 在屏幕上显示该图。 你也可以使用 `savefig` 将绘图保存在文件中。

`pyplot` 的文档位于 http://matplotlib.sourceforge.net/ 。

练习 6

测试 `LinearMap` ， `BetterMap` 和 `HashMap` 的性能；看看你能否描述它们的增长级别。

你可以从 `thinkcomplex.com/Map.py` 下载我的映射实现，以及从 `thinkcomplex.com/listsum.py` 下载我在本节中使用的代码。

你必须找到一个 `n` 的范围，它大到足以显示渐近行为，但小到足以快速运行。

附录 B、阅读列表

> 原文：Appendix B Reading list
>
> 译者：飞龙
>
>
>
> 自豪地采用谷歌翻译

以下是一些精选的书籍，介绍复杂性科学，并且是了解该领域的全貌的一种愉快的方式。

- Axelrod, Complexity of Cooperation.
- Axelrod, The Evolution of Cooperation.
- Bak, How Nature Works.
- Barabasi, Linked.
- Buchanan, Nexus.
- Epstein and Axtell, Growing Artificial Societies: Social Science from the Bottom Up.
- Fisher, The Perfect Swarm.
- Flake, The Computational Beauty of Nature.
- Goodwin, How the Leopard Changed Its Spots.
- Holland, Hidden Order.
- Johnson, Emergence.
- Kelly, Out of Control.
- Kluger, Simplexity.
- Levy, Artificial Life.
- Lewin, Complexity: Life at the Edge of Chaos.
- Mitchell, Complexity: A Guided Tour.
- Mitchell Waldrop: Complexity, the emerging science at the edge of order and chaos.
- Resnick, Turtles, Termites, and Traffic Jams.
- Rucker, The Lifebox, The Seashell, and the Soul.
- Sawyer, Social Emergence: Societies As Complex Systems.
- Schelling, Micromotives and Macrobehaviors.
- Schiff, Cellular Automata: A Discrete View of the World.
- Strogatz, Sync.
- Watts, Six Degrees.
- Wolfram, A New Kind Of Science.